VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Simulace prostředí pro projekt Jiný Kosmos

Environment Simulation for Project Other Cosmos

2018 Bc. Dominik Čech

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Dominik Čech**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Simulace prostředí pro projekt Jiný Kosmos
Environment Simulation for Project Other Cosmos

Jazyk vypracování: čeština

Zásady pro vypracování:

Práce přímo navazuje na vývoj simulačního prostředí Jiný Kosmos, ketrý probíhá v rámci semestrálních projektů. Cílem práce je vytvořit simulační mechanizmus pro dynamické části světa a entity, který bude využívat vlákna a umožní tak využít vícejádrové procesory. Dalším optimalizačním mechanizmem bude snížení četnosti simulačních kroků pro nedůležité objekty v případě přetížení simulačního procesu. Dalším simulačním mechanizmem bude naplánování vyvolání určité akce po uplynutí stanovené časové prodlevy pro entity jejihž simulační stav se mění v delších časových úsecích.
Simulační mechanizmus umožní:
1. Vícevláknovou simulaci.
2. Optimalizaci časových kroků pro méně důležité entity v případě přetížení.
3. Vyvolání akce po uplynutí specifikované doby.
Práce bude obsahovat:
1. Přehled používaných metod a algoritmů.
2. Implementaci výše popsané funkcionality.
3. Experimenty a vyhodnocení výsledků s ohledem na výkon.
4. Programátorskou dokumentaci řešení s využitím diagramů jazyka UML.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] NUTARO, James. Building software for simulation: theory and algorithms, with applications in C++. Hoboken, NJ: Wiley, c2011.

Dále dle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

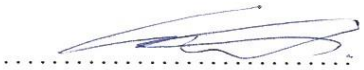
Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2018

.....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 4. května 2018



Rád bych na tomto místě poděkoval Ing. Davidu Ježkovi, Ph.D. za rady a podněty, které přispěly k dokončení této diplomové práce

Abstrakt

Cílem práce byla implementace simulačního mechanismu pro různé entity v herním prostředí projektu Jiný Kosmos. Celý mechanismus měl být naprogramován s ohledem na výkon, rozšiřitelnost systému a možností simulace stovek až tisíců entit v daném světě. Simulační systém ve hře je plně funkční a připraven na další rozšíření. V práci popisuju jak teorii a příklady použití simulací, tak implementaci daného mechanismu.

Klíčová slova: Java, LibGDX, Jiný Kosmos, simulace

Abstract

The goal of this work was to implement simulation mechanism for various objects in project called Other Cosmos. Simulation mechanism was implemented with scalebility and performance in mind. There is posibility to simulate hundreds to thousands objects at once. Simulation mechanism is fully functional and ready to be extended. In my paper, you can find some theory and examples about usage of simulations in real world and some implementation details of simulation mechanism.

Key Words: Java, LibGDX, Other Cosmos, simulation

Obsah

Seznam použitých zkratek a symbolů

DES	–	Discrete-event simulation
GUI	–	Graphical user interface
UML	–	Unified Modeling Language
FPS	–	Frames per second
CPU	–	Central Processing Unit
RAM	–	Random access memory
OpenGL	–	Open Graphic Library
WAV	–	Waveform Audio File Format
MP3	–	MPEG-1/MPEG-2 Audio Layer 3

Seznam obrázků

1 Úvod

Tato práce přímo navazuje a rozšiřuje semestrální projekt „Jiný Kosmos“. Tento projekt si dává za cíl vytvořit digitální svět, ve kterém se hráč může volně pohybovat po okolí a utvářet tento svět podle své libosti. Celý projekt „Jiný Kosmos“ byl od záčátku spoluprací několika studentů, včetně mě, kde jsme vytvořili procedurálně generovaný herní svět tvořen bloky ve stylu známé úspěšné hry „Minecraft“. Takový svět by však nebyl příliš zábavný pokud by nenabízel interaktivní prvky jako tekoucí vodu nebo možnost kombinovat jednotlivé logické stavební bloky do větších a komplexnějších výtvorů. Komplexnějším výtvorem může být například vytvoření elektrických obvodů, hradel a dalších součástek pro vytvoření jednoduchého počítače uvnitř samotného herního světa. Aby toto bylo možné, je třeba navrhnout a implementovat systém umožňující simulovat stovky až tisíce stavebních bloků se svou vlastní logikou a prezentovat výstupy těchto bloků hráči.

Projekt „Jiný Kosmos“ je složením několika náročných částí jako samotné vykreslování, procedurální generování terénu, stromů, řek a dalších objektů, až po simulaci chování herních entit a bloků, čímž se zabývá tato práce. V první části práce se pokusím přiblížit, co to simulace je, na jaké typy se rozděluje, k čemu jí lze využít a kde na ní můžeme narazit. Dále se zaměřím na samotný návrh a implementaci, popsání problémů, na které jsem během vývoje narazil, a optimalizací, které byly zapotřebí pro plynulý běh celé hry.

2 Simulace

Simulace je synonymem pro imitování reálných jevů, systémů nebo procesů. Abychom mohli něco simulovat je v první řadě třeba vytvořit model, který představuje klíčové vlastnosti, chování nebo funkce daného reálného fyzického jevu, systému nebo procesu. Model reprezentuje samotný systém, přičemž simulace představuje průběh daného systému, jevu či procesu v čase.[1]

Simulace je široce používaná v různých odvětvích jako automobilový průmysl, ekonomie, počasí, sport, finance, vesmír, počítačové hry a další. Také se používá pro vědecké modelování přírodních jevů a procesů, kde pomáhá objasnit jejich fungování. Simulací také můžeme ukázat reálné výsledky různých přístupů k řešení nějakého problému a dané výsledky následně porovnat. Simulace se také často používají na systémy, které například ještě ani neexistují nebo jsou potencionálně nebezpečné, nedostupné.[1]

Proč využíváme simulace v tak velkém měřítku a rozsahu, jsem již trochu přiblížil v předešlém odstavci. Mezi hlavní důvody použití je možnost zjistit, jak se daný systém bude chovat v různých podmínkách aniž bychom museli systém fyzicky vytvořit. Toto se může například hodit při zjišťování jak se bude chovat raketa při přistání na Marsu. Zjištění této skutečnosti by přinejmenším bylo velice náročné bez použití počítačové simulace. Počítačová simulace nám v tomto ohledu dává možnost vytvořit přibližný fyzikální model atmosféry Marsu, který poté můžeme použít pro simulování jednotlivých přístání rakety, aniž bychom zbytečně zničili jedinou raketu. Další využití počítačové simulace může být v odvětví výuky letání nebo chirurgie, kde nám jednotlivé modely nahradí reálné letadlo či operační sál. Mezi výhody také jistě patří možnost zrychlit simulaci jevů, které trvají delší dobu nebo naopak zpomalit ty, které probíhají rychle.

Hlavní nevýhodou simulací je potřeba vytvořit daný model systému. Problém není ani tak vytvořit model systému, ale vytvořit ho správně a co nejpřesněji, aby byl vhodný pro použití k predikci reálného systému.

2.1 Klasifikace simulací

Fyzická simulace Jedná se o simulaci, kde fyzické objekty jsou náhradou za reálné objekty systému z důvodu ceny nebo menších rozměrů.

Interaktivní simulace Speciální druh fyzické simulace, která navíc zahrnuje lidského operátora.

Deterministická simulace Simulace, kde jednotlivé proměnné závisí na deterministický algoritmu, tím pádem opakované spouštění simulace se stejnými krajními podmínkami má stejný výsledek.

Stochastická simulace Simulace, kde jednotlivé proměnné jsou odvozeny na základě Monte Carlo techniky používající pseudo náhodná čísla, tím pádem spouštění simulace se stejnými krajními podmínkami končí jiným výsledkem v rozmezí intervalu spolehlivosti.

Spojitá simulace Simulace, kde se jednotlivé proměnné mění spojitě v čase.

Diskrétní simulace Simulace, kde se jednotlivé proměnné mění skokově, například na základě událostí.

Počítačová simulace Simulace běžící na počítači.

2.2 Počítačová simulace

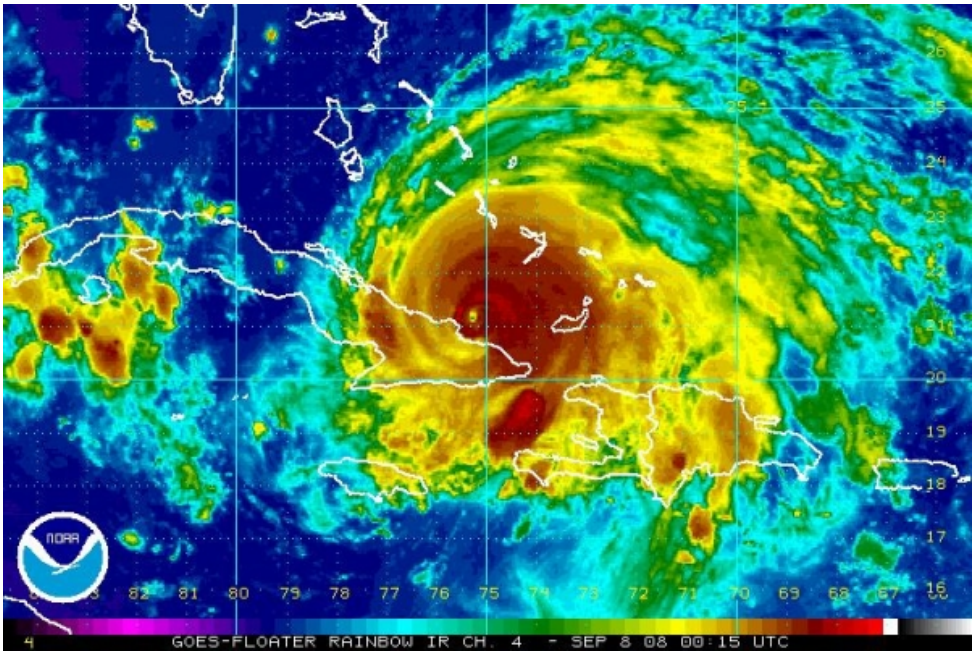
Počítačová simulace je pokus o modelování reálných nebo hypotetických situací na počítači. Co nás k tomuto vede, je možnost prostudovat daný systém z různých pohledů a zjistit jak přesně funguje. Změnou jednotlivých parametrů simulace můžeme předpovídat chování daného systému. Počítačovou simulaci můžeme využít jako virtuální nástroj pro zjištění chování systému, který studujeme[1].

Tento typ simulace se velice široce používá pro modelování přírodních jevů z fyziky, chemie nebo biologie. Dalším zajímavou aplikací počítačových simulací je, když počítač simuluje činnost jiného počítače. Tomuto typu simulátoru se běžně říka emulátor. Tohoto se často využívá v situacích, kdy potřebujeme zjistit, jak se bude chovat nějaký program na jistém zařízení aniž bychom dané zařízení měli fyzicky k dispozici. Příkladem by mohl být vývoj aplikace pro iOS nebo Android zařízení. Ikdyž jsou tyto zařízení v celku běžné ve většině domácnostech, při samotném vývoji je velice časově náročné vždy danou aplikaci nahrávat na cílové zařízení a kontrolovat, jak funguje. Místo toho můžeme využít emulátor, který nám dané zařízení nahradí přímo na naší vývojové stanici. Dalším důvodem použití emulátoru může být z důvodu, že zařízení buď už neexistuje nebo je velice vzácné/drahé. Například staré hry vytvořené pro první konzole si dnes můžeme díky emulátorům zahrát i dnes.

Bez počítačových simulací by nebyla možná ani předpověď počasí. Dnešní moderní simulace počasí dokáží varovat 15 až 20 minut před nebezpečnou bouří nebo tornádem, také v celku dobře předpovídají počasí na dva až tři dny dopředu[13].

Způsob, jak je toto možné, je provedení simulace, která představuje všechny naše znalosti o dynamice kapalin a atmosférických podmínkách, nad velkým množsví dat poskytnutých z nejrůznějších meteorologických stanic, radarů a satelitů. Tyto simulace většinou běží na světových superpočítačích, které i přes jejich rychlost dokáží pouze předpovídat počasí pro plochu až pět čtverečních mil[13]. Na obrázku 1 můžeme vidět výsledek jedné z takových simulací.

Některé vědecké teorie dokonce naznačují, že život, jak ho známe, by mohl být pouhým výsledkem velice komplexní simulace.



Obrázek 1: Předpověď počasí pomocí pořítačové simulace[12]

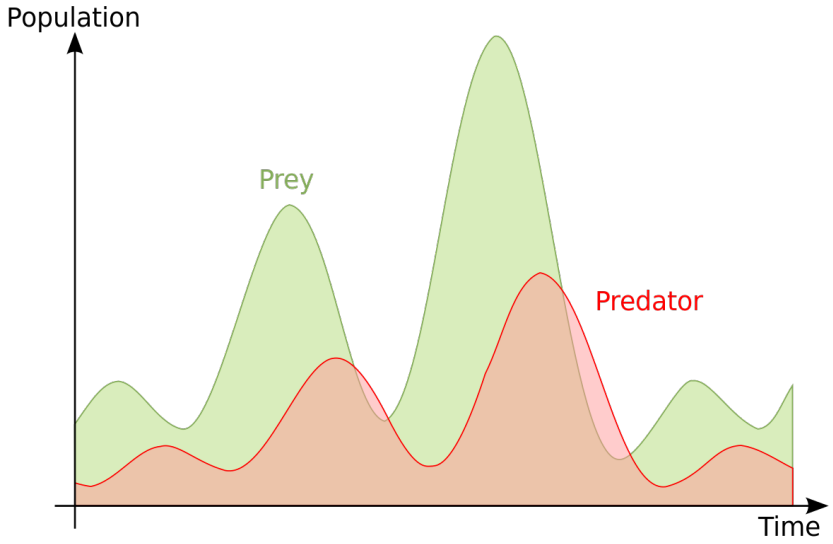
2.3 Spojitá simulace

Spojitá simulace se zabývá modelováním systému v průběhu času, ve které se stav proměnných mění průběžně v závislosti na čase[2]. Jako příklad takové simulace si můžeme představit třeba měření teploty nebo rychlost padajícího tělesa. Typicky se pro reprezentaci simulace používá množina diferenciálních rovnic, které nám poskytují vzájemný vztah četností změn stavu v čase[2]. Pro vyřešení těchto rovnic se využívají dvě metody.

- Analytická
- Numerická

Množině diferenciálních rovnic se také říká koncepční model, který abstraktně popisuje daný systém. Příkladem jednoduchého koncepčního modelu může být například model predátora a kořisti. Tento model popisuje dva typy organizmů, v našem příkladu zvířat, predátora a kořist. Tyto dvě populace zvířat představují jednoduchý potravní řetězec, kde predátoři loví kořist a kořist se pase na vegetaci. Velikost těchto dvou populací zvířat lze vyjádřit dvěmi diferenciálními rovnicemi, které představují vztah mezi jednotlivými populacemi a jak se velikost jednotlivých populací navzájem ovlivňuje.

Z obrázku 2 jde vidět, že velikost populace predátorů reaguje s malým spožděním na velikost populace kořisti. S malou populací predátorů začne růst populace kořisti. V reakci na to, s malým spožděním, začné růst i populace predátorů, což má však za následek pokles populace kořisti a tím pádem i pokles populace predátorů. Poté se celý cyklus opakuje.



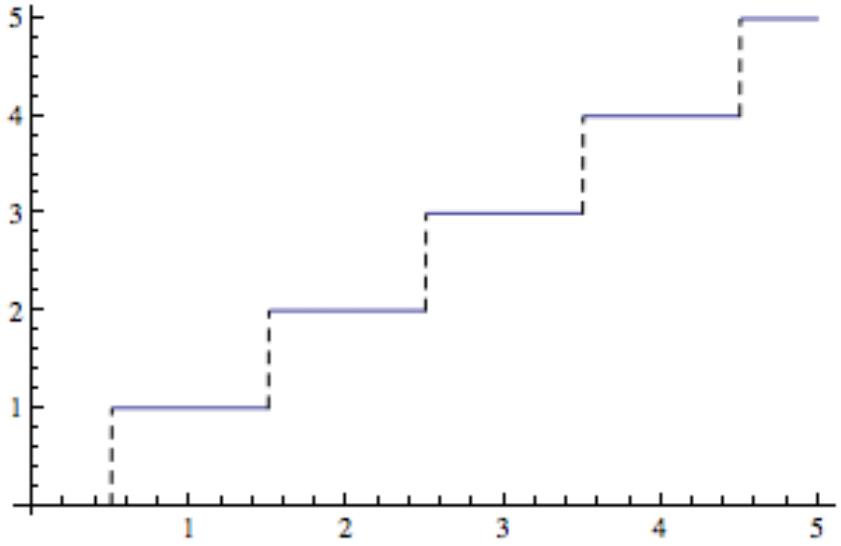
Obrázek 2: Křivky výsledku simulace modelu predátora a kořisti dle Lotka–Volterrových rovnic

Tento typ simulace má nekonečně mnoho stavů a lze jí nahradit diskretní simulací, kde výsledek takové diskretní simulace je aproximace výsledku nahrazené spojité simulace. Převod na diskretní simulaci lze například udělat pomocí metody vzorkování, kde zjišťujeme stav simulace v určitých krocích.

2.4 Diskrétní simulace

Typ simulace je také znám pod zkratkou DES. Jedná se o simulaci, která je charakteristická tím, že se systém modeluje jako diskrétní sekvence událostí v čase. Jednotlivé události nastávají za sebou a každá mění stav systému, přičemž se předpokládá, že mezi událostmi nenastává žádná změna stavu systému. Jedním z typických příkladů pro diskrétní simulaci, může být například internetový obchod a vliv objednávek na skladové zásoby. V takovém případě by jedna z událostí byla příchozí objednávka, tato událost by snižovala skladové zásoby produktu, což by byl náš stav systému. Jelikož skladové zásoby je třeba doplňovat, další událost by navyšovala skladové zásoby produktu. Kdybychom takovou simulaci spustili a události generovaly v náhodném časy, vznikl by nám graf reprezentující skokovou funkci množství skladových zásob v čase.

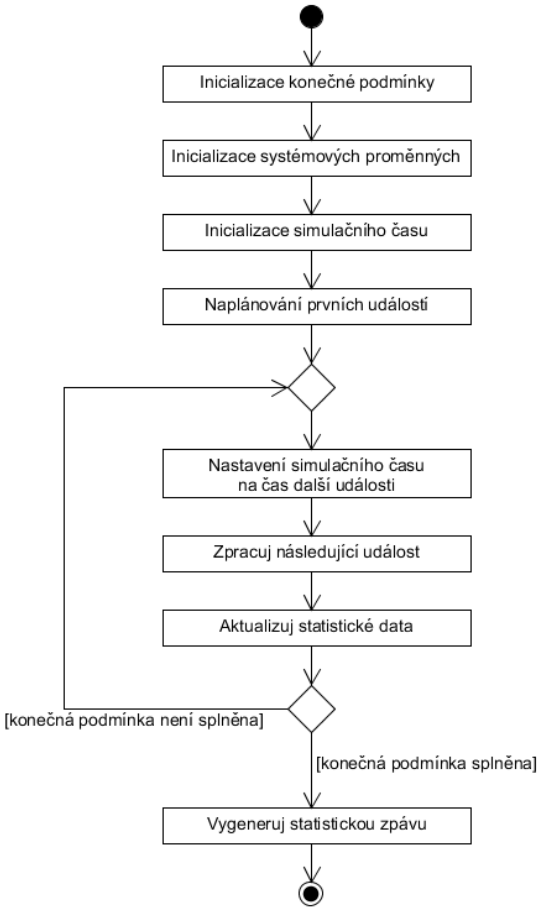
Samotné programování simulací může být celkem náročný úkol, složitý na napsání kódu až po ladění výsledné simulace. Hlavním důvodem je paralelní programování, kdy běží mnoho událostí v jeden okamžik, což stěžuje obecný náhled na to, co program dělá. Z tohoto důvodu lidé vymysleli různé simulační jazyky nebo alespoň simulační programátorské paradigmata, která pomáhají v pochopení simulačního kódu. SIMULA je jedním z jazyků, který byl vyvinut v 60 letech speciálně pro psaní simulačního kódu[3]. Tento jazyk se stal také významným z důvodu představení objektově orientovaného programování, které je dnes tak široce používané. Každopádně dnešním trendem je spíše vytvářet knihovny zaměřené na simulace, které lze spouštět z běžných jazyků jako je například C++, místo vytváření celých nových jazyků. [3]



Obrázek 3: Graf skokové funkce[5]

Každá diskrétní simulace kromě logiky reagující na jednotlivé události, ještě většinou obsahuje tyto části:

- Stav systému tvořící množinu proměnných, které daný systém popisují. Stav v čase lze také matematicky vyjádřit skokovou funkcí, která mění svou hodnotu v závislosti na příchozích událostech.
- Předdefinovaný začátek a konec simulace, což může být například některá z událostí nebo časová konstanta.
- Metoda měření času od začátku simulace. Metoda měření se může měnit v závislosti na zvolené implementaci simulace. V některých případech můžeme mít konstatní simulační krok, v jiných případech se může simulační čas skokově měnit podle příchozích událostí.
- Událostní fronta je nedílnou součástí každe diskrétní simulace. Každá simulace si vede frontu událostí, které je potřeba vykonat. Tato fronta se plní na základě předem vykonaných událostí. Občas je také vhodné si pamatovat události, které jsme již provedli.[4]
- Událost samotná by měla obsahovat čas, kdy nastala, a typ. Podle typu lze poté určit jaká logika se pro danou událost spustí. Občas také může obsahovat dodatečné parametry pro potřeby provedení logiky dané události. Některé události můžou také kromě počátečního času uvádět informaci, kdy daná událost končí. V tomto případě je událost spuštěna minimálně dvakrát, případně i s jinou logikou.
- Každá simulace by si měla vést statistiky průběhu simulace daného modelu. Tato data jsou poté prezentována uživateli pro vyvození závěru.



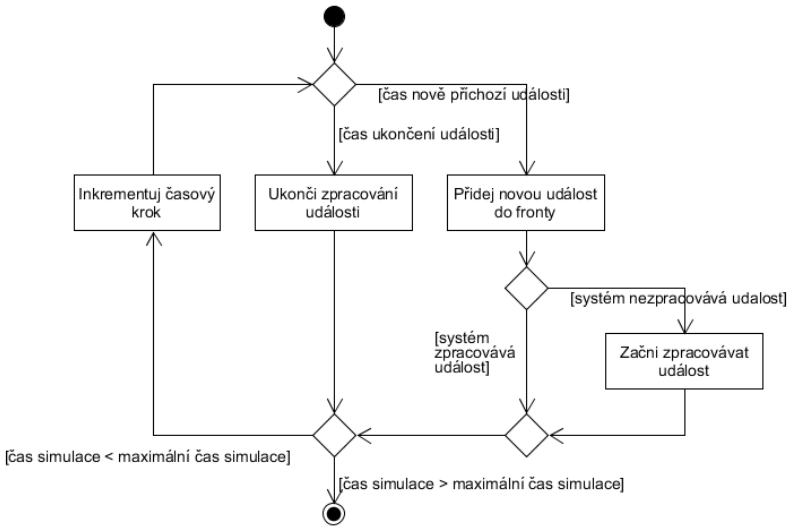
Obrázek 4: UML aktivitní diagram řídící logiky simulace

Obecné fungování řídícího kódu simulace můžeme vidět na obrázku 4. V první řadě je třeba správně nastavit konečnou podmínku, prvotní stav systému, čas začátku simulace a naplánovat první události. Poté se již ve smyčce aktualizuje čas simulace podle simulačního časového kroku nebo začátku další události, načtení nové události a smazání staré z fronty událostí a aktualizace statistických dat do doby, než dojde k ukončení pomocí konečné podmínky. Po ukončení simulační smyčky dojde ke zpracování všech statistických dat a případně k vygenerování zprávy o simulaci.

V dalších částech se pokusím shrnout a popsat několik paradigmat DES programování, jejich výhody a v čem se navzájem liší.

2.4.1 Aktivitně orientované paradigma

V aktivitně orientovaném paradigmatu se čas rozdělí na malé časové skoky. Například pokud chceme simulovat čas odbavení zákazníků u pokladny a víme, že mezi jednotlivými příchody zákazníka k pokladně je 30 sekund, potom si můžeme zvolit časový krok simulace například



Obrázek 5: Hlavní simulační smyčka aktivitně orientovaného paradigmatu

0.5 sekundy[3]. V těchto intervalech náš kód bude kontrolovat možné nové příchozí zákazníky(aktivity) nebo se starat o zákazníky(aktivity) ve frontě. Každému novému zákazníku je vygenerovaná doba jeho odbavení a dalé se generuje doba, za jakou příjde další zákazník. Pokud se naplní doba odbavení zákazníka, který je na řadě, tak je odstraněn z fronty a následuje další zákazník. Po uplynutí dané simulační doby můžeme zjistit výsledky simulace, což v tomto příkladu může být průměrná doba odbavení jednoho zákazníka a počet odbavených zákazníku v daném časovém intervalu. Doba mezi jednotlivými příchody zákazníků a doba odbavení zákazníků u pokladny by měla být spojitá náhodná veličina dle některé z distribučních funkcí. [3]

Nevýhodou v tomto přístupu je hlavně doba nutná k provedení simulace, jelikož kontrolujeme stav co určený časový krok. Toto může vést ke zbytečným spouštěním simulačního cyklu, kde nedojde ke změně stavu systému, což v našem příkladu znaméná žádný nový příchozí zákazník nebo odbavení zákazníka. V případech, kde simulace běží i několik dní, například u výrobců elektronických součástek, kteří používají DES pro simulaci čipů, by tato metoda nebyla velice vhodná.[3] Jinou variantu nabízí událostně orientované paradigma, ve kterém se tomuto problému můžeme vyhnout.

2.4.2 Událostně orientované paradigma

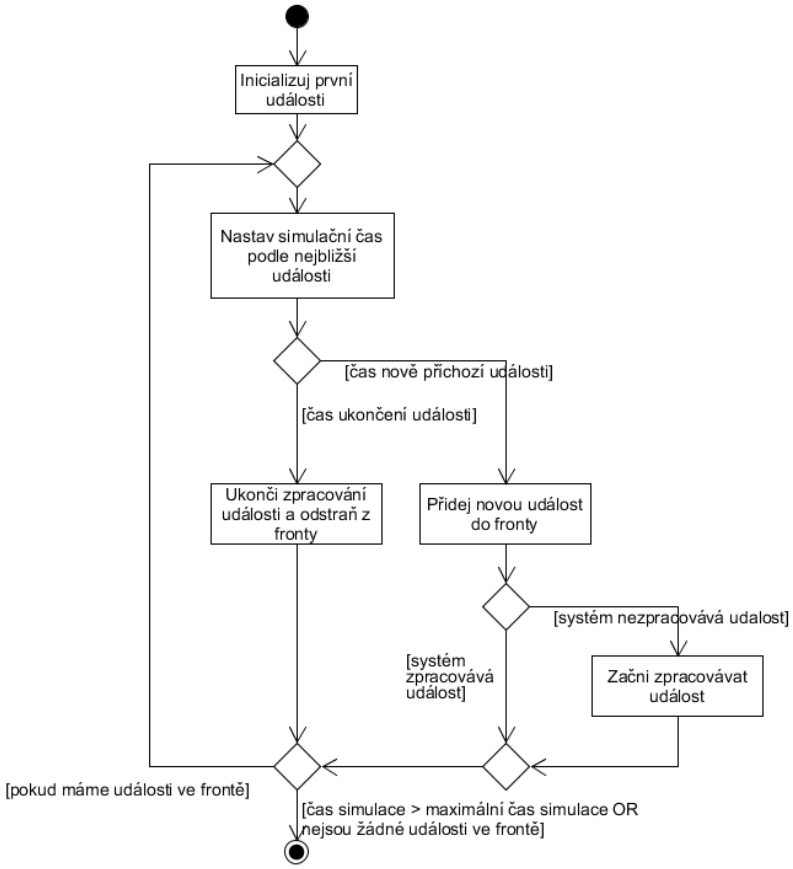
Hlavním problémem v aktivitně orientovaném paradigmatu bylo plýtvání časem na simulační cykly, které neměnily stav systému. Toto paradigma však funguje jinak a to místo toho, aby simulace byla posunována po určeném časovém kroku, posunujeme se po jednotlivých událostech[3]. Abychom tohoto dosáhli, musíme mít frontu událostí, která nám bude určovat náš postup. Fun-

gování simulace pomocí tohoto paradigmatu se pokusím popsat podle upraveného příkladu, uvedeného v kapitole 2.4.1 Aktivitně orientované paradigma.

V prvotním průchodu simulačním cyklem máme prázdnou frontu a dojde k vygenerování nového zákazníka(události), kterému se poznačí doba příchodu, dle simulačního času, a délka doby odbavení. Také dojde k určení příchodu dalšího zákazníka(události). V dalším kroku se zkontroluje fronta zákazníků(událostí) a simulační čas se nastaví na hodnotu podle nejbližší události. V případě, že čas příchodu nového zákazníka je menší než čas odbavení stávajícího zákazníka, nastaví se čas na hodnotu příchodu nového zákazníka jinak na čas odbavení stávajícího zákazníka. Toto představuje jednotlivé skoky mezi událostmi. Poté se celý proces opakuje a podle toho na jakou hodnotu je nastaven náš simulační čas, můžeme poznat jestli máme přidat nového zákazníka do fronty nebo odstranit odbaveného zákazníka. Koncovou podmínkou simulace může být například předem stanovená časová doba simulace.

Důležitou podmínkou je, že musíme vybírat události postupně podle času, jak jdou za sebou. Toto však nemusí odpovídat skutečnému pořadí v naší datové struktůře. V případě, že máme systém kde dochází k hodně událostem, je vhodné použít datovou struktůru jako například prioritní frontu. V hodně případech se však používá fronta jako klasický spojový list, tato struktůra je vhodná, pokud počet událostí vyskytujících se v systému, není moc velký.

Mezi hlavní výhody tohoto přístupu je celkem jednoduchá implementace simulačního cyklu a samozřejmě rychlost provedení simulace.



Obrázek 6: Hlavní simulační smyčka událostně orientovaného paradigmatu

2.4.3 Procesně orientované paradigma

Tento typ paradigmatu je podobný událostně orientovanému paradigmatu s tím rozdílem, že jednotlivé aktivity simulace jsou spuštěny paralelně ve více vláknech. Aktivity simulace by v našem příkladu byly zaznamenání příchozího zákazníka a odbavení zákazníka. Tyto dvě aktivity by běžely paralelně a byly by řízeny hlavním vláknem, které by se staralo o probouzení jednotlivých aktivitních vláken podle událostí ve frontě a toho kdo je do fronty zařadil.

Výhodou tohoto přístupu je jasné rozdělení simulace do několika na sobě nezávislých celků, které jsou řízeny hlavním řídícím vláknem, a komunikují spolu přes událostní frontu. Logika hlavního řídícího vlákno a přepínání mezi jednotlivými aktivitami je většinou součástí simulačních knihoven. Hlavní náplní programátora je tedy poté napsat samotnou logiku simulace. Výsledný kód je poté většinou lépe čitelný a srozumitelnější. Z tohoto důvodu je toto paradigma více populárnější než aktivitní nebo událostně orientované paradigma.

2.5 Simulace v počítačových hrách

Většina počítačových her jsou vlastně jakési simulace nějakého světa se svými pravidly, a pokud samotná celá hra není jednou velkou simulací, tak minimálně obsahuje menší prvky, které simulace využívají. Zde bych rád představil pár projektů, kde simulace je základním herním prvkem.

2.5.1 Factorio

Hra, která svým vzhledem možná nevypadá nejlépe, ale díky svému propracovanému systému sběru a zpracování surovinných zdrojů, se nyní řadí mezí velice úspěšné a oblíbené hry v komunitě hráčů.

Obrázek 7: Ukázka ze hry Factorio[8]

Ve hře zaujímáte pozici inženýra, který ztroskotal na neznámé planetě plné surového bohatství a vaším úkolem je na této planetě přežít. K tomu, aby jste přežili budete muset začít stavět stroje na těžbu surovin, pomocí kterých můžete vyvíjet nové technologie a stroje, které vám pomůžou v přežití. Celý systém těžby a zpracování je nakonec změť automatizovaných strojů a pásů, které si jednotlivé suroviny a výrobky předávají a produkují další, komplikovanější vynálezy. Aby toho nebylo málo na planetě se skrývají tvorové, kteří nejsou nadšení z vašeho plundrování planety a snaží se vám vaši továrnu zničit, proto je třeba vytvářet i zbraňové systémy. Tyto systémy však k vaší obraně potřebují munici, kterou samozřejmě musíte vyrobit. Jedná se tedy o celkem zajímavý projek, kde vaším úkolem je postavit co nejefektivnější výrobní linku na nejrůznější výtvory. Na obrázku 7 můžete vidět pár náhledů jak jednotlivé továrny vypadají přímo ve hře.

2.5.2 Kingdom Come: Deliverance

Česká hra od studia Warhorse Studio si dala za cíl vytvořit hru ze 14.století v době před husitskými válkami. Jejich hlavním cílem bylo vytvořit hru, která by co nejvěrněji zachycovala danou dobu. V této hře každá entita jak postava tak zvířata, mají naprogramované chování jejich denodenního cyklu. Nejedná se o nějaké jednoduché naskriptované chování, ale reálnou simulaci, kdy to, co daná postava bude dělat, se odvíjí od stavu jeho vlastností jako je hlad, žízeň, vyčerpání, povolání a další. Daný denní cyklus se poté ještě odvíjí od interakce s hráčem. Postavy také umí různě na hráče reagovat a to tak, že když hráč dělá něco podzřelého můžou například zavolat strážné, aby danou situaci prověřili. Postavy jsou simulovány i v případě, že hráč není v jejich okolí a to z důvodu toho, aby stav simulace jednotlivých postav byl korektní a nestávaly se situace, kdy hráč příjde uprostřed noci na pole a uvidí tam pracujícího sedláka, který by místo toho měl být v posteli nebo hospodě. Obrázek 8 ukazuje grafické pojetí celé hry.

Obrázek 8: Ukázka ze hry Kingdom Come: Deliverance[9]

Další zajimavou části hry je simulace boje, která se neodehrává klasickým zběsilým klikáním myši, ale strategickým vyčkáváním a využitím menší staminy oponenta. Při šermu se fyzika jednotlivých mečů simuluje a tím pádem jednotlivé nárazy na brnění jsou ve skutečnosti nárazy a né pouhé promáchnutí skrz oponenta. Tato skutečnost má potom vliv na stav brnění, které má hrač na sobě. Na obrázku 9, můžete vidět jak takový souboj vypadá. Hra také vyniká svou nádhernou grafikou.

Obrázek 9: Ukázka šermu[9]

3 Simulace v projektu Jiný Kosmos

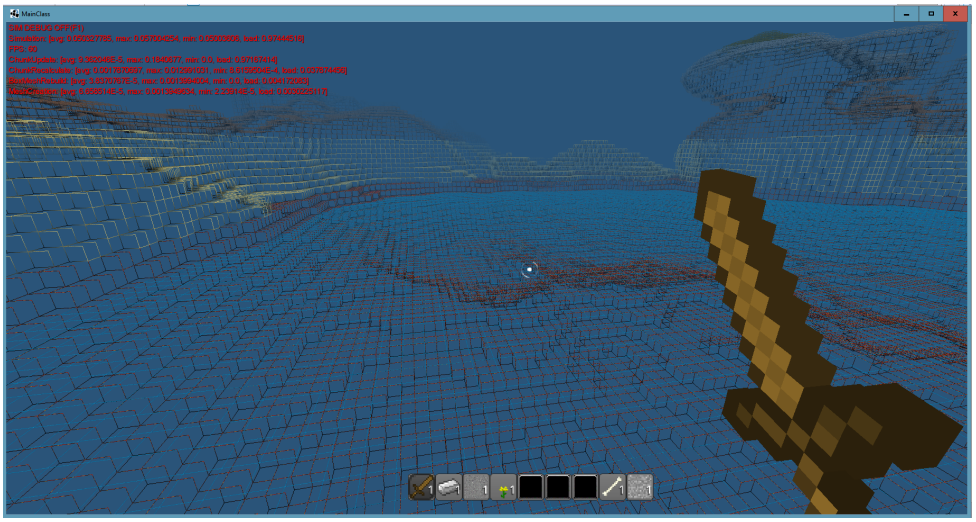
Jak jsem již napsal v kapitole [1] Úvod, projekt „Jiný Kosmos“ je svým návrhem a vzhledem podobný hře Minecraft. Rád bych tedy v pár větách obecně shrnul jak celá hra funguje. Celý svět je tvořen jednotlivými bloky. Každý blok má definované své vlastnosti, jako například jak vypadá, jestli umi uchovávat svůj stav, jestli má nějaké chování(simulaci), která se spouští za určitých podmínek. Skupina bloků o rozměrech 16x16x256(šířka, hloubka, výška) poté tvoří jeden takzvaný chunk. Tento chunk je ještě z optimalizačních důvodů virtuálně rozdělen na menší sektory. Jednotlivé pozice bloků a typy se poté generují zcela procedurálně v reálném čase podle toho jak se hráč pohybuje. Teoreticky je tedy možné běžet v jednom směru do nekonečna a nikdy nenarazíte na stejnou krajinu jako předtím.

Jakékoliv změny terénu, které hráč nebo simulace udělají, se počítají v ramci jednoho sektoru daného chunku. To znamená, že pokud odeberu jeden blok z vykresleného světa, musí se projít všechny bloky daného sektoru a přepočítat vzhled terénu, který se následně uloží ve formě meshe a pošle se do vykreslovací sběrnice OpenGL. Jak vypadá terén ve formě meshe, můžete vidět na obrázku 10, můžete si povyšimnout, že jediné co se vykresluje, je povrch, ale ne co je pod ním. Toto je z důvodu vykreslovací optimalizace, kdy není potřeba vykreslovat to co hráč nevidí. O celý proces vykreslování se stará knihovna LibGdx, která představuje vrstvu nad OpenGL a celý vývoj poněkud usnadnila. Celý proces aktualizace terénu není tedy časově úplně triviální záležitostí. V případech, kdy terén upravuje hráč, je vše relativně v pořádku, protože jednotlivé změny terénu hráčem jsou relativně pomalé a tím pádem vše se stíhá překreslovat rychle. V okamžiku kdy změny dělá simulace, jelikož se jedná o desítky až stovky změn, které se nemusí podařit zpracovávat najednou, celý proces překreslování může celou hru zpomalit až do bodu, kdy není hratelná. Toto byl jeden z problémů, které bylo třeba vyřešit pro bloky s chováním, které dělají hodně změn terénu.

Implementace simulací nebyla tedy pouze o naimplementování mechanismu, který by byl schopný provádět logiku jednotlivých bloků a entit světa, ale také úprava již existujících systémů jako například vykreslování.

Celý mechanismus simulací je navržen tak, aby nebyl závislý na zbylých systémech hry. Tímto je myšleno, že v budoucnu je možné oddělit simulační část od zbylých systému hry a nahradit například komplexní vykreslování hry za jednoduché GUI, ve kterém se můžou například testovat jednotlivé simulační skripty pro bloky. Také se v budoucnu může hra rozšířit pro mód s více hráči, kdy bude potřeba některé části mít na straně serveru a některé zase na straně klienta. Proto vývoj celého systému simulací byl vyvíjen tímto způsobem. Simulační část má tedy jasně definované rozhraní, pomocí kterého komunikuje s ostatními systémy hry. Rozhraní celého simulačního mechanismu můžete vidět v kapitole [4] Programátorská dokumentace.

Vývoj celého mechanismu simulací a úpravy ostatních systémů nebyla jednorázova implementace. Vzhledem k tomu, že jsem nikdy nic podobného neprogramoval, jednalo se spíše o



Obrázek 10: Terén vykreslen v podobě meshe

různé pokusy, až jsem se dostal do finální podoby. Celý vývoj se pokusím přiblížit v pár dalších kapitolách této práce.

3.1 Úpravy ostatních systémů hry

K tomu, aby bylo možné doimplementovat mechanismus simulací do hry, bylo nezbytně nutné upravit již stávající systémy hry. Jednalo se především o proces aktualizace jednotlivých chunků, který byl důvodem propadů FPS při vyšším počtu simulovaných bloků.

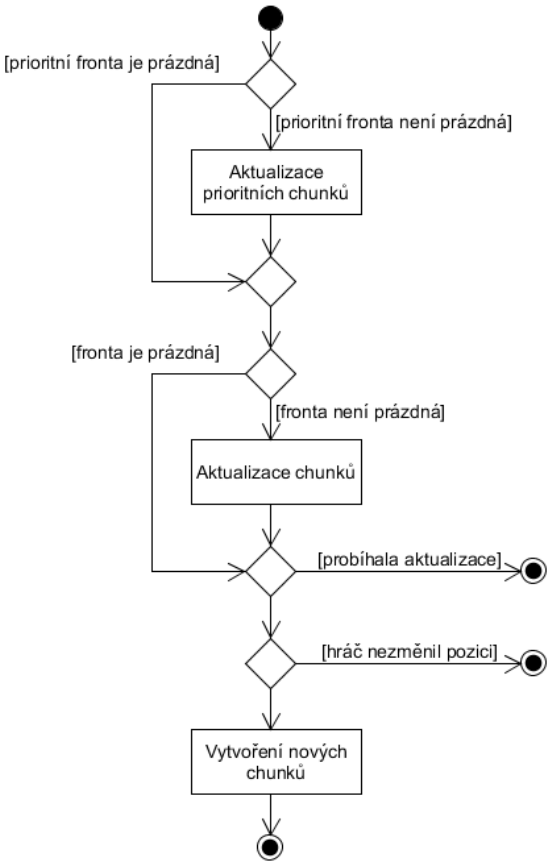
Další částí, která byla nutná doimplementovat, byla podpora stavu u bloku. V této části bylo třeba zajisti, aby si blok mohl uchovávat svůj stav. Do této doby to nebylo možné jelikož vždy existuje pouze jedna instance bloku ve hře a jednotlivé bloky v chunku jsou spojeny s touto instancí pomocí unikátního ID. Tento způsob reprezentace byl hlavně zvolen kvůli paměťovým nárokům. Úkladání stavu bloku bylo velice důležité doimplementovat, jelikož některé simulace si potřebují ukládat informace o daném stavu. Například se jedná o blok vody, u které je vhodné si ukládat úroveň hladiny. Tato hodnota se potom promítá do samotného vykreslování, kdy můžeme vidět jak se hladina vody mění.

Úpravám se nevyhnulo ani vykreslování terénu. Zde se spíše jednalo o změny ve struktůře kódu, aby byl lépe čitelný a podporoval změny v aktualizaci chunků.

3.1.1 Aktualizace chunků

Na obrázku 11 lze vidět diagram jak vypadala aktualizace chunku před úpravou. V podstatě se jednalo o kontrolu dvou front a to fronty prioritních chunků a obecné fronty chunků. Tyto dvě fronty sloužily k určení, jaké chunky je třeba aktualizovat, případně prioritně aktualizovat. Samotná aktualizace potom probíhala ve vytvořeném novém vlákně, které po aktualizaci všech chunků ve frontě zaniklo. Nelze úplně říct, že vlákno po aktualizaci zaniklo, jelikož všechna

vlákna byla vybírána z takzvaného thread pool. V podstatě se pouze jedná o předpřipravený list několika vláken, která čekají na využití. Vytvoření instance vlákna je poměrně náročná operace a neustálé vytváření nových vláken by bylo neefektivní. I přes použití thread pool, nebyl tento způsob úplně ideální. Hlavně tedy z důvodu problému se synchronizací jednotlivých vláken. Dalším problém bylo například spouštění celého vlákna pouze pro aktualizaci pár chunků. Navíc k tomu byl celý kód zanesen různými zámky a synchronizačními bloky a byl problém se v něm zorientovat. Pokud obě aktualizační fronty byly prázdné, mohlo se přistoupit ke kontrole jednotlivých existujích chunků a případné naplánování aktualizace chunku. V případě, že by hráč změnil pozici a to tak, že by se pohl z jednoho chunku do jiného, bylo potřeba ještě vytvořit nové chunky. Tímto způsobem fungovala stará verze aktualizace chunků, většina věcí v nové verzi zůstala podobně až na použití thread pool.



Obrázek 11: Aktivitní diagram aktivace chunků

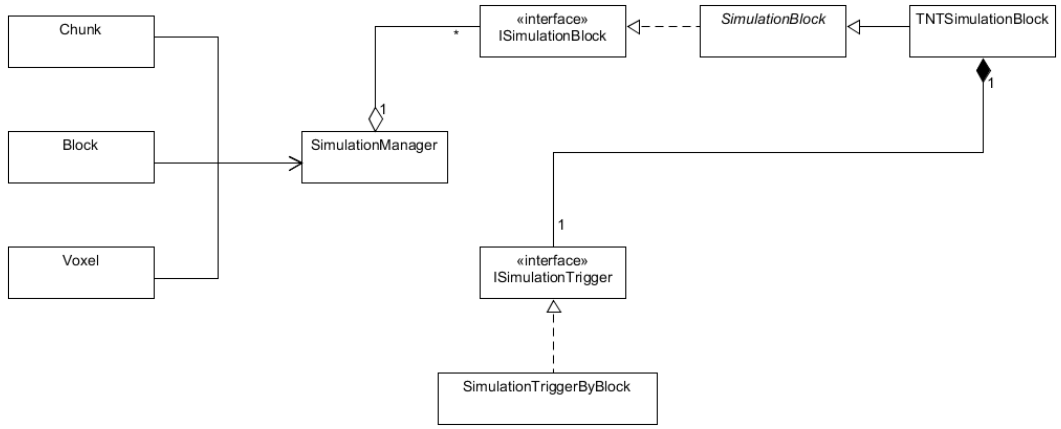
Hlavní změnou je, že jsem místo thread pool použil jedno vlákno, které se inicializuje při vytvoření herního světa. Toto vlákno se poté stará o samotnou aktualizaci chunků a běží tak dlohou dokud je hra zapnutá. Dále jsem místo klasické fronty, kterou v předešlé implementaci představoval spojový seznam, použil blokující spojový seznam. Tento seznam dokáže zablokovat

vlákno v případě, že je prázdný a v okamžiku kdy do něho něco vložíme, znovu vlákno probudí. Tato datová struktůra mi dala pěkný způsob jak ovládat aktualizační vlákno z hlavního vlákna. Takový způsob komunikace mezi vlákny je jedním z návrhových vzorů, který se nazývá Producer Consumer. Výhodou v tomto přístupu vidím jasnější rozdělení logiky jednotlivých vláken a dále okamžitou aktualizaci chunku v momentu, kdy chunk vložím do aktualizační fronty.

3.2 Koncept simulačního procesu

Tato kapitola by měla v krátkosti představit koncept simulačního procesu, který byl implementován před dnešním simulačním procesem a v mnoha ohledech se od sebe liší. Obrázek 12 představuje třídní diagram starého simulačního procesu. Na tomto třídním diagramu můžeme vidět, že celý systém je závislý na několika ostatních systémech hry, kde tyto vazby byly v novém systému simulačního procesu odstraněny.

Na to, aby bylo možné vytvořit novou simulaci, je nejprve nutné mít blok, který má simulační logiku. Toto se provede při specifikaci nového bloku ve třídě BlockTypes, kde jedním z parametru je třída, ze které lze vytvořit dané chování. Daná třída samozřejme nemůže být jakákoliv, ale musí implementovat rozhraní ISimulationBlock. Daný block má potom metodu, která pomocí této specifikované třídy umí vytvořit instanci simulace. Tímto způsobem je dosaženo spojení konkrétního bloku s konkrétní simulací. Zde najdeme další rozdíl od nového návrhu, kde blok implementuje rozhraní, které lze přidat do simulačního procesu.



Obrázek 12: Třídní diagram koncepční implementace simulačního procesu

Při změnách terénu je třeba zajistit, aby se jednotlivé bloky v daném chunku zkontrolovaly, jestli se nemá spustit jejich simulační logika z důvodu provedené změny. Tohoto je docíleno voláním metody kontroly bloku v simulačním manažeru na každý blok v chunku při jeho přepočítávání. Každé přepočítání chunku je vyvoláno při změně jeho struktůry, čímž je zajištěno, že vždy dojde ke kontrole všech simulačních bloků. V novém návrhu je toto odstraněno a o samotné zařazení bloku do simulačního procesu se stará sám programátor.

Aby došlo k samotnému spuštění jednotlivých simulací ve frontě, je třeba zavolat aktualizační metodu simulačního manažeru. Ta se volá ze třídy Voxel. Při čemž dojde k průchodu listu simulací a k jejich spuštění. Každá simulace poté může vygenerovat změny terénu, které se posbírají a po odsimulování všech simulací, dojde k jejich aplikování. Velkým rozdílem oproti novému návrhu je ten, že celý simulační proces běží synchronně, takže nevznikají problémy s aplikováním změn a nemusí se používat složitější datové struktůry pro reprezentaci fronty.

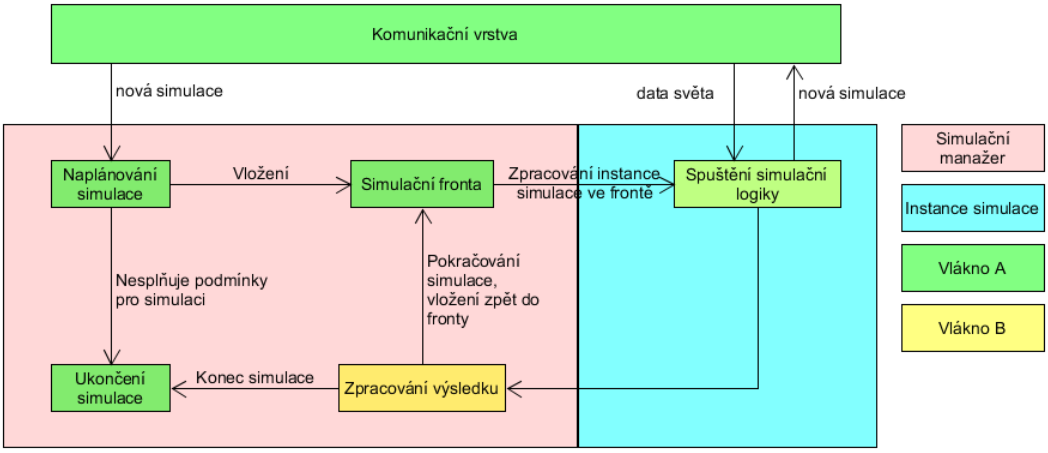
Třídy typu ISimulationTrigger sloužily pouze jako pomocné třídy pro spouštění daných simulací. Například vznikla třída SimulationTriggerByBlock, která umožňovala jednoduše v dané simulaci nastavit jaké bloky budou simulaci spouštět.

3.3 Simulační proces

V této kapitole bych nejprve rád obecně prošel, jak celý simulační proces funguje a poté bych se podíval na jednotlivé detaily implementace tohoto procesu a jaké postupy jsem při implementaci vyzkoušel. Na konci kapitoly bych potom rád srovnal dva různé přístupy v implementaci simulačního procesu.

Pro jednodušší pochopení jsem přiložil obrázek 13, který celý proces popisuje. V okamžiku, kdy chceme zahájit simulaci daného bloku, musíme jí naplánovat. Při naplánování simulační manažer zkontroluje instanci daného bloku a pokud se jedná o blok, který obsahuje nějaký druh chování, pokusí se blok přidat do simulačního cyklu. Simulační manažer si uchovává informaci o simulovaných blocích a to ve formě hashe, který je vytvořen pomocí pozice bloku a jeho typu nebo-li unikátního ID bloku. Díky tomuto je zajištěno, že daný blok na dané pozici nebude do simulace umístěn více než jednou. V případě, že blok projde všemi nutnými podmínkami, je vložen do simulační fronty. Fronta je postupně zpracovavána a pro každý blok se spouští jeho simulace. Jednotlivá logika simulace může generovat změny světa a plánovat simulace pro další bloky. Po každém dokončení simulace bloku se vrací výsledek do simulačního manažeru. Jedná se o stav, který rozhodne o tom jestli je třeba pokračovat další iterací simulace nebo je možno simulaci bloku odstranit. Kromě pokračování a ukončení simulace lze také určit/změnit s jakou prioritou se má simulace spustit nebo pro speciální typy simulace lze nastavit uspání po nějakou dobu.

V krátkosti jsem tedy popsal životní cyklus simulace a teď bych se rád podíval na jednotlivé detaily. Samotný proces zpracování simulační fronty běží paralelně v jiném vlákně. Z diagramu lze vidět, že do fronty může vkládat jak logika simulace, která běží ve vlákně B tak i vlákno A, v kterém vložení do fronty může vyvolat například hráč. Kdybychom použili klasický spojový list museli bychom složitě hlídat přístupy do datové struktůry z jednotlivých vláken. Tento přístup se mi nezdál nejlepší proto jsem zvolil dvojitý spojový list, kde jeden list je striktně pro čtení a druhý pro zápis. Takže pokud vlákno B iteruje jedním z listů, tak se všechny nově plánované simualce vkládají do listu druhého. V okamžiku kdy vlákno B doiteruje přes všechny simulace v jeho daném listu, předá se kontrola zpět simulačnímu manažeru, který listy prohodí. Tím pádem vlákno B dostane nový plný list. Z tohoto důvodu je třeba, aby každá simulace vracela



Obrázek 13: Diagram simulačního procesu

svůj výsledek se stavem. Pokud má simulace nastaveno, že má pokračovat, simulační manažer musí danou instanci přesunout do druhého listu. Dvojitý spojový list vyřešil kolize ve čtení a zápisu dvou vláken najednou a navíc mi poskytnul dobrou kontrolu nad procesem řízení simulací v simulační smyčce.

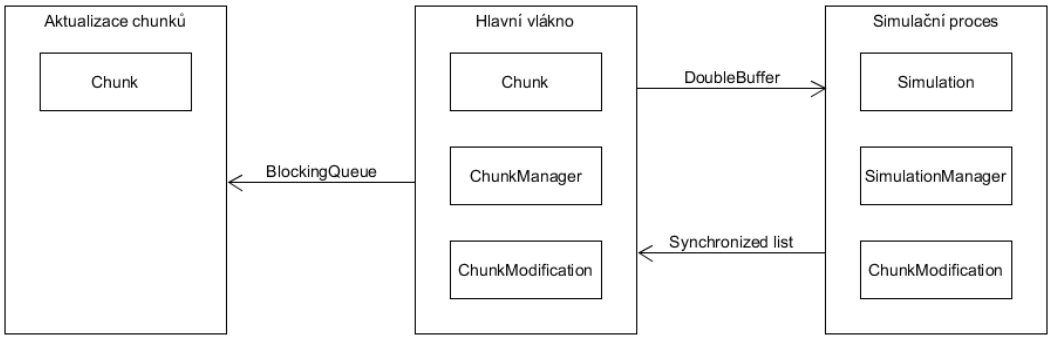
V předešlém odstavci jsem zmínil, že celý proces zpracování jednotlivých simulací ve frontě probíhá v jiném vlákně. I zde, stejně jako v kapitole [3.1.1] Aktualizace chunků, jsem problém řešil dvěmi způsoby, kde jsem se nakonec rozhodl pro samostatné vlákno oproti thread pool variantě.

- Thread pool varianta fungovala způsobem, kde hlavní vlákno dokola volalo aktualizační metodu simulačního manažeru a pokud zrovna neprobíhala simulace a bylo co simulovat, vytvořilo se nové vlákno pro obsluhu simulační fronty. Problémem v této variantě byla kontrola jak často se jednotlivé simulace spouští, nebo-li dodržet takzvaný simulační krok.
- Druhá varianta byla vytvořit jedno simulační vlákno při inicializaci světa a uspávat ho podle potřeby. Tuto variantu jsem nakonec zvolil právě kvůli možnosti kontroly simulačního kroku, kdy pokud čas odsimulování je menší než můj zadaný simulační krok tak dané vlákno uspím na dobu rozdílu v časech. Dalším důvodem je, že mi tento způsob usnadnil vytvoření takzvaného vyvážení zatížení jako jednu z optimalizačních technik, kterou popíšu v kapitole [3.6] Optimalizace

3.4 Vlákna a komunikace mezi nimi

Celý systém operuje se třemi paralelně běžícími vlákny a to:

Hlavní vlákno Jedná se o hlavní a zároveň vykreslovací vlákno. Zároveň kromě vykreslování také slouží k celkové inicializaci hry a k aktualizacím různých systému jako je fyzika a jiné. Také spouští zbylé dva vlákna.



Obrázek 14: Jednotlivé vlákna a jejich komunikace

Simulační proces Vlákno jehož hlavní prioritou je spouštět jednotlivé simulace.

Aktualizace chunku Toto vlákno se stará pouze o přepočítávání změn jednotlivých chunků.

Hlavní vlákno je jádrem celého systému a ostatní vlákna od něj dostávají příkazy kdy mají co dělat. Mezi hlavním vláknem a vláknem aktualizace chunků probíhá jednosměrná komunikace a to od hlavního vlákna. Komunikace je na bázi blokující fronty, kdy vlákno aktualizace chunků čeká až do doby než se ve frontě objeví nějaky objekt typu Chunk. Poté dojde k jeho zpracování, ale hlavní vlákno se již nestará, kdy k jeho zpracování došlo, pouze že ho vložilo do fronty.

Komunikace mezi hlavním a simulačním vláknem je obousměrná. Hlavní vlákno může vytvářet požadavky na odsimulování simulačních bloků či entit. Tyto objekty se musí vložit do simulační fronty, která je zpracovávána simulačním vláknem. V tomto případě je třeba zajistit, aby si jednotlivé vlákna neměnily navzájem frontu. Toho je docíleno použitím návrhového vzoru DoubleBuffer popsané v této kapitole [3.11.1]. Zpětná komunikace s hlavním vláknem je ve formě modifikací terénu, které simulace během svého vykonávání udělaly. Tyto objekty modifikací se zapisují k jednotlivým chunkům, na kterých byly provedeny, a poté jsou hromadně aplikovány. V tomto případě je použit jednoduchý synchronizovaný list, který zajišťuje v jednom okamžiku jeho čtení nebo zápis do něj.

3.5 Propagace změn a vykreslování

Jedním z velkých problémů byla propagace změn tvořené logikou simulací ze simulačního vlákna zpět do hlavního vlákna jednotlivých chunků.

Prvním a trochu naivním způsobem propagace byla přímá úprava vzhledu chunku ze simulační logiky. Toto mělo za následek hned několik problémů. Jeden z problémů byla samotná synchronizace aktualizačního vlákna chunku a samotného simulačního vlákna, kdy jedno vlákno chtělo upravovat datové struktůry chunku a druhé vlákno potřebovalo číst tyto struktůry. To by nebyl až tak velký problém, ale díky tomu vznikaly při vykreslování různé artefakty, jako bloky vody na místech, kde již neměly být. Dalším problémem byla frekvence přepočítávání terénu.

Chunk se nastaví na aktualizaci hned při první změně, která se u něho zaregistruje a poté nastane přepočítání terénu a vykreslení nového terénu daného chunku. Jak jsem již v dřívější části podotknul přepočítání a znovu vykreslení nové podoby terénu není časově triviální záležitost a v případech jako simulace propagace vody, nastává změn u chunků až v řádech desítek tisíc. Takový počet rychlých změn za sebou vede k zahlcení aktualizačního vlákna chunků a začne docházet k trhaným vizuálním změnám.

Druhý způsob byl již o něco lepší a to takový, že místo aplikování změn rovnou, jsem si jednotlivé změny ukládal k danému chunku, ale mimo hlavní datovou struktůru terénu chunku. Každá nová změna se k chunku uloží do listu změn k aplikování, jako nová instance objektu ChunkModification, která obsahuje pozici změny a typ bloku, který se má na danou pozici nastavit. Změny se poté reálně aplikují předtím, než se začne přepočítávat terén chunku. Samotné přepočítání chunku se poté musí vynutit po odsimulování simulační fronty. Tento postup vedl k méně vykreslovacím artefaktům a k omezení trhání vizualizace.

Třetí způsob pouze trochu vylepšuje předešlou snahu o propagaci změn a to tím, že se vynucení přepočítávání chunků nedělá po odsimulování simulační fronty, ale po určených časových intervalech. Tímto se modifikace simulací akumulují a až poté se aplikují. Interval vynucení přepočítání je určen tak, aby aktualizační vlákno mělo dost času přepočítat všechny potřebné chunky.

Celý proces vizualizace změn simulací podstoupil hodně změn, které vedly k velkému navýšení výkonu vykreslování, ale je zde pořád dost prostoru pro zlepšování. Počet modifikací, které vznikají při některých simulací je v řadech desítek tisíců až stovek tisíců. I když jsem nepozoroval žádné velké propady výkonu kvůli instanciování tolika objektů, ne úplně se mi tento postup líbí a to hlavně z důvodu toho, že bylo potřeba upravit čtení dat z chunku. V okamžiku, kdy se čtou data z chunku jako například jaký blok je na pozici x, y, z, není možné rovnou vrátit hodnotu uloženou v datové struktůře, ale je nutné se první podívat jestli na dané pozici nebyla vytvořena modifikace. Toto se musí kontrolovat z důvodu poskytnutí aktuálních dat. Bohužel mě nenapadl jiný šetrnější postup, jak pro CPU nebo paměť RAM, který by řešil problémy výše popsané.

3.6 Optimalizace

Pro lepší běh vyššího počtu simulací byly přidány některé optimalizační metody. Tyto optimalizace jsou obecného typu, které může využít jakýkoliv typ simulační logiky. U některých typů simulací i toto však není dost pro pěkný plynulý běh aplikace a bylo by třeba optimalizovat danou simulaci na míru. Například propagace vody, u tohoto typu simulace se předpokládá velké množství instancí pohybujicích se bloků vody. Přičemž samotné bloky vody jsou součástí takzvaného meshe chunku. Mesh je v podstatě výsledkem přepočítání terénu chunku a představuje jakousi vrstvu, kterou hráč vidí. To však má za následek, že v okamžiku kdy se změní pozice nebo hladina bloku vody, musí se vždy přepočítat celý kus meshe chunku. Jedna z takových optimalizací přímo na míru danému simulačnímu typu by mohla v tomto případě být, rozdělení meshe samotného terénu chunku a meshe vody. Kdybychom toto udělali, přepočítávali

bychom pouze mesh vody a tím odlehčili práci pro přepočítávání chunku. Toto však v dosavadní implementaci není.

3.6.1 Prioritní zpracování

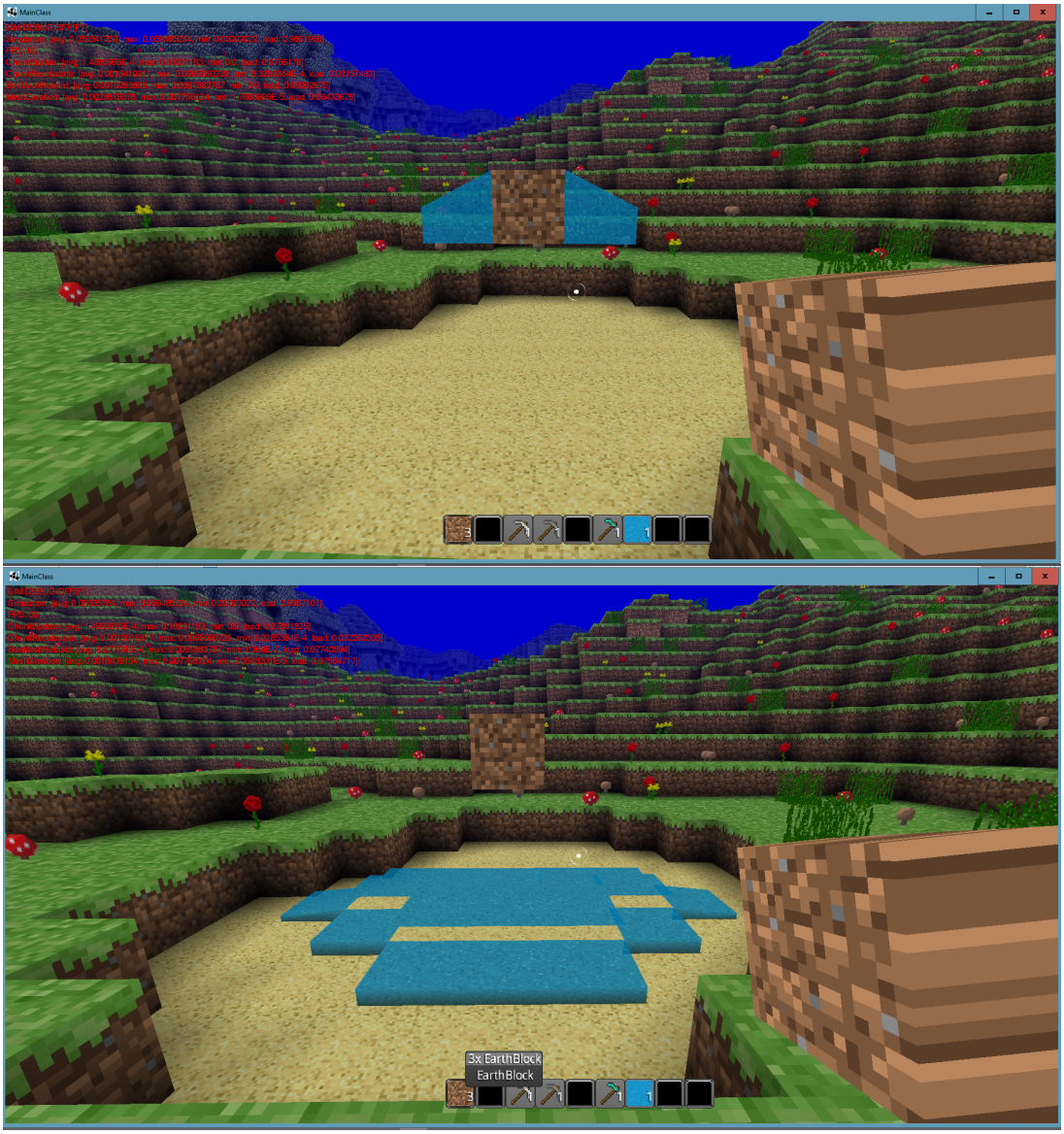
V nynější době existují tři úrovně priority a to „vysoká“, „střední“, „normální“ a „nízká“. Každá z těchto priorit má nastaven časový interval, ve kterém se musí daná simulace spustit. Jednotlivé úrovně priorit jsou interně implementovány pomocí dvojitého spojového listu, který jsem popisoval v kapitole 3.3 Simulační proces. Při průběhu simulačním cyklem se průběžne provádí simulace v jednotlivých listech od nejvyšší priority až po nejnižšší. Simulace s nejvyšší prioritou se provádí vždy, ale další úrovně se provádí pouze v případě, že čas strávený prováděním předešlých simulací není větší než nastavený simulační krok. Pokud se však neprovedou simulace s nižší prioritou delší dobu než je u dané priority nastavený interval, simulace se automaticky vynutí sama. V případě přetížení simulačního vlákna to znamená, že simulace s vysokou prioritou se spouští vždy a nižší priority jsou přeskakovány určitý počet simulačních cyklů podle toho o jakou úroveň priority se jedná.

Výchozí nastavení priority při naplánování simulace je na normální úroveň. Toto však lze změnit buďto poskytnutím prioritního parametru ve funkci scheduleSimulation nebo v logice samotné simulace při nastavování, jestli daná simulace má pokračovat v dalším simulačním cyklu.

3.7 Časové simulace

Typ simulace, který lze uspávat a znovu spouštět. Celý proces uspávání a probouzení řeší instance časového plánovače(TimeScheduler). Díky tomuto lze měnit systém úspávání za běhu simulační logiky. V nynější době jsem implementoval pouze jeden takový plánovač a to krokový. Krokový časový plánovač si jako parametr při vytvoření bere čas po jaké době se má daná simulace uspat a poté probudit. Také lze určit kolikrát se má daná simulace spustit nebo jak dlouho má celkově běžet než se odstraní. Tímto způsobem jsem dosáhl dynamického přizpůsobení běhu dané simulace. V komplexnosti plánovače se meze nekladou a záléží pouze na programátorovi, jak složité chování naimplementuje.

Tento typ simulace můžete vidět v praxi na obrázku 15, na kterém lze vidět jeden ze simulačních bloků, který co 5 sekund vytváří bloky vody, které poté padají k zemi, kde se rozpustí.



Obrázek 15: Zachycení bloku vytvářející vodu co 5 sekund

3.8 Chování entit

Entity narozdíl od bloků jsou dynamické objekty ve světě. Co tím myslím je to, že každá entita je ve světě reprezentovaná svou vlastní instancí a vykreslována zvlášť. To umožňuje například rychlou aktualizace polohy dané entity. Na druhou stranu jednotlivé bloky nejsou reprezentovány jako celky, ale až vetší skupina sousedicích bloků je takto reprezentována. Tím pádem při změně pozice jednoho bloku je třeba celý celek aktualizovat.

Entity se z tohoto důvodu plánují pro simulaci trochu jinak. Každá entita, která by měla být simulovaná na simulačním vlákně, musí rozšiřovat třídu ActorBehaviour. Tato třída implementuje všechny potřebné rozhraní, které jsou potřeba, aby mohla být přidána do simulační fronty.

Jedním z rozdílů je ten, že samotná metoda simulate, která obsahuje simulační logiku pro danou entitu, nemusí vracet výsledek simulace. Minimálně ne přímo, jelikož třída ActorBehaviour se o toto stará sama. Důvodem tohoto je fakt, že při simulaci entity lze předpokládat běh po celou dobu života entity. V případě, že chceme přidat entitu s chováním do světa, musíme použít metodu addActor ve třídě World, která zařídí jak přidání do simulační fronty tak samotné vykreslení entity.

Obrázek 16: Chyba ve vykreslování

Kromě těchto pár detailů simulace entit fungují naprosto stejně jako simulace bloků. Při testování většího počtu entit jsem narazil na zvláštní chybu, pravděpodobně ve vykreslovacím vlákně, kteoru můžete vidět na obrázku 16. Bohužel se mi nepodařilo přijít na přícinu této chyby. Chyba nevzniká vždy, ale nahodile. Je možné, že chyba má co dočinění s pokusem vykreslit entitu v době kdy už byla odstraněna z paměti a poté dojde k rozhození celého vykreslování. Tento problém jsem však pozoroval jenom v případech kdy jsem měl vykreslen velký počet entit a byly široce rozprostřeny po celém světě.

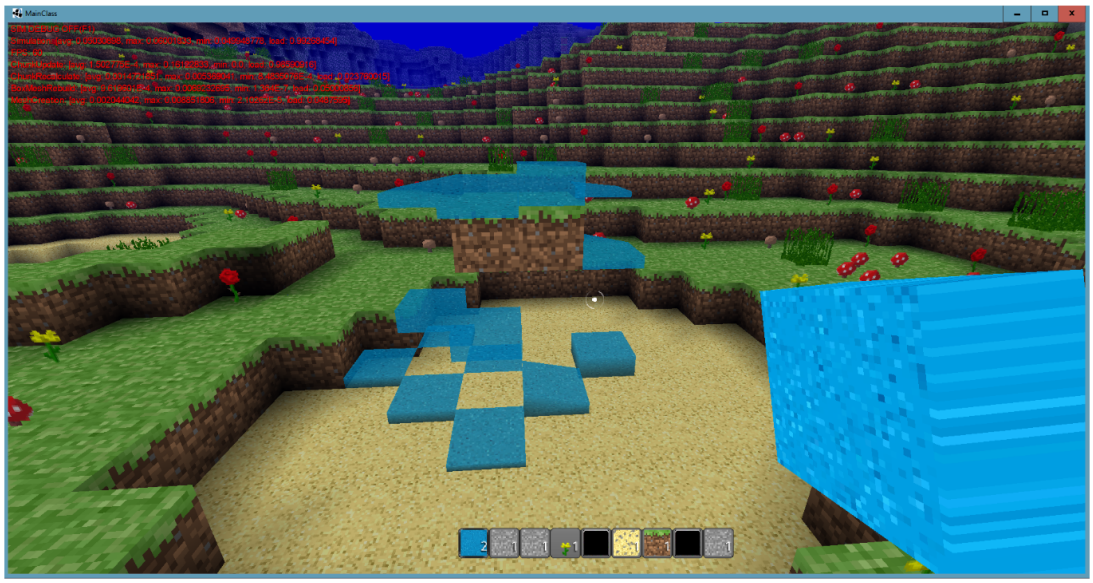
3.9 Propagace vody

Logika propagace vody je hlavní simulace, na které jsem celý systém testoval. Výsledná simulace není úplně nejlepší a určitě by šla ještě dále zlepšit, jelikož cílem práce bylo naprogramovat mechanismu pro simulování a ne jednotlivé simulace bloků. Na testování však postačila dobře. Celá logika je postavená na modelu Cellular Automaton.

Cellular Automaton je model tvořený systémem buněk, které mají specifické chrakteristiky.

- Buňky existují v mřížce. V našem případě se jedná o tří dimenzionální mřížku, ale obecně mřížka může mít jakýkoliv konečný počet dimenzí.

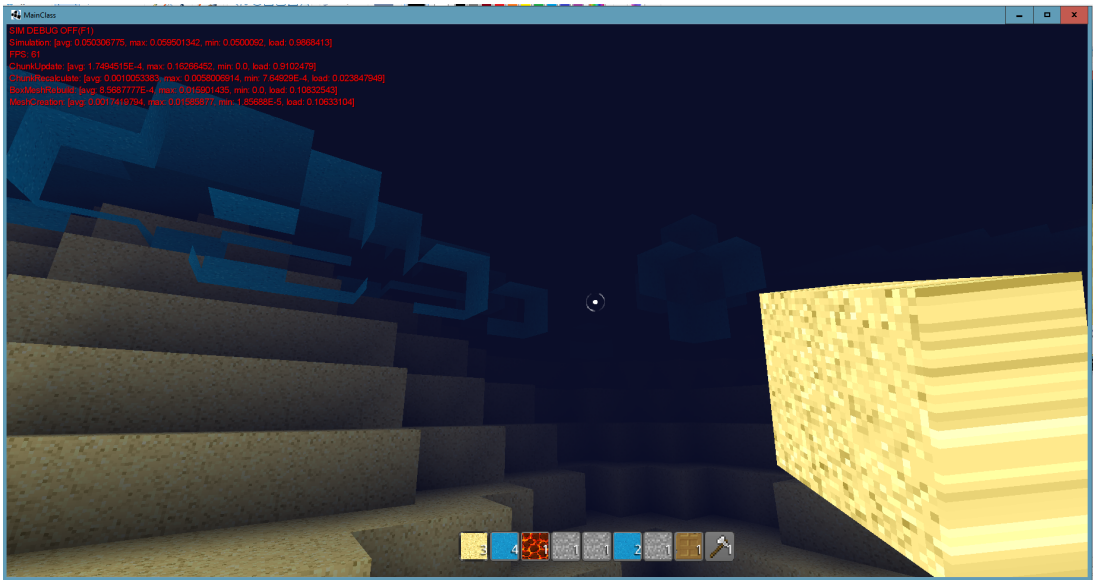
- Každá buňka má svůj stav. V našem případě se jedná o stav, který nám říká kolik daná buňka obsahuje vody.
- Každá buňka má své okolní buňky.



Obrázek 17: Náraz vody na pevný objekt

Z tohoto listu a z podstaty jak naše hra funguje, můžeme říct, že všechny předpoklady splňujeme. Výpočet nové hodnoty buňky se určuje dle stavu dané buňky a jejich sousedících buněk z minulosti. Tento způsob je velice elegatním řešením jak právě například „jednoduše“ simulovat kapaliny. Tento model také používá algoritmus na šíření světla v herním světě. S celým model přišel Stanisław Ulam a John Von Neumann [6]. Asi nejznámější aplikací Cellular Atomaton je takzvaná Hra života.

Obrázek 17 ukazuje jak funguje simulace vody při nárazu na pevný objekt. Na druhou stranu obrázek 18 ukazuje jak vypadá simulace vody pod její hladinou, lze si všimnout, že se pod hladinou vytváří bubliny, ty však bohužel zůstávají na místě a nevyvěrávají na hladinu.



Obrázek 18: Simulace vody pod hladinou

3.10 Porovnání výkonosti v různých fázích vývoje

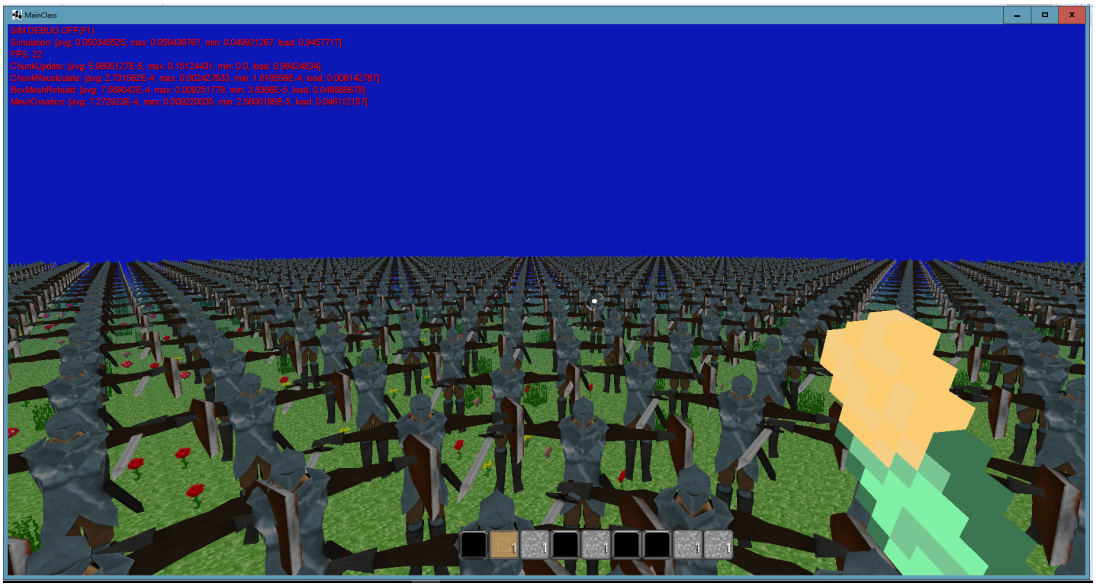
V této kapitole bych rád věnoval pár řádků pro porovnání jednotlivých verzí implementace. Jak jsem se již několikrát zmínil, některé části systému jsem přepisoval několikrát a vždy jsem se snažil daný systém zlepšit a zrychlit. Ne vždy tomu však tak bylo a některé změny neměly na výkon takový efekt, který bych si představoval. Porovnání rozdělím na tři části, u kterých vždy napíšu jak daná implementace vypadala a jak si vedla při simulaci vody.

První fáze Tato fáze vývoje byla ve stavu kdy samotná simulační smyčka byla implementována pomocí thread pool varianty a modifikace světa byly aplikovány rovnou na datové struktůry chunků. Také nedocházelo k žádnému zpomalování vykreslování změn. V této fázi se mi reálně dařilo simulovat okolo 800 vodních bloků, ale svět začínal byt nerezponzivní. FPS spadlo na hranici 40.

Druhá fáze Zde je simulační smyčka stále implementována pomocí thread pool varianty, ale modifikace se již aplikují až po dosimulování celé simulační fronty. Tento krok celkem zrychlil celou simulaci a podařilo se mi dostat až na 2000 vodních bloků, kdy svět začal být nerezponzivní. FPS spadlo na hranici 51.

Třetí fáze V této fázi došlo ke kompletnímu přepsání vykreslování chunků a simulační smyčky, která místo thread pool varianty používá jedno stále běžící vlákno. Tyto změny však neměli tížený efekt a výsledky simulace vodních bloků zůstaly na stejných hodnotách.

Při těchto pokusech jsem si všiml, že není ani tak problém v rychlosti provádění velkého počtu simulací, ale spíše v jejich vizualizaci. Aktualizační vlákno chunku jednoduše nestíhá rychle



Obrázek 19: Simulace 3600 entit

přepočítávat jednotlivé změny, které simulace provádí a potom trhaný vizuální výsledek navádí k tomu, že problém je v samotném počtu simulovaných bloků. Toto však není pravda, jelikož tím že vše běží v rozdílných vláknech, pohyb hráče se ve většině případech nezačne zasekávát, tím pádem nespadne ani FPS. FPS hodnota padá pouze v případech kdy se vykresluje velké množství objektů a již samotné vykreslovací vlákno přestává stíhat. Pokud však pozorujeme problém s responzivitou světa nebo pomalým nahráváním nových chunků, určitě se jedná o přetížené aktualizační vlákno chunků.

Po dokončení implementace simulace entit, jsem také zkoušel kolik entit dokaže systém snést. Při testech jsem se dostal na 3600 entit, které simulovali pouze pohyb dané entity. FPS již spadlo na hodnoty kolem 19, přičemž samotná simulační smyčka stále stíhala aktualizovat jednotlivé entity co 50 milisekund. Jedná se o další příklad kde simulační systém by zvládl i více entit, ale je v tomto případě limitován vykreslovacím vláknem. Jak vypadal test 3600 entit můžete vidět na obrázku 19

Všechny testy byly prováděny na stolním počítači s těmito paramtery:

- Intel Core i5-7400 taktován na 3GHz
- 8GB paměť RAM
- GeForce GTX 1060 - 3GB
- Windows 10 - 64bit

3.11 Návrhové vzory

Tato kapitola se bude zabývat několika návrhovými vzory, které jsem v projektu použil. Kromě popisu daného návrhového vzoru také napíšu, kde je daný vzor vhodné použít a jak jsem ho využil já.

Návrhových vzorů existuje celá řada a znalost některých z nich je vždy výhodou. Pomáhají v našem kódu řešit situace, které už byly řešeny před námi a to metodami, které podporují udržitelnost a rozšířitelnost našeho kódu do budoucna. Také je si ale třeba uvědomit, že ne vždy je vhodné vše implementovat přesně tak jak daný návrhový vzor popisuje, v některých případech je třeba použít selský rozum a zbytečně věci nepřekombinovat. I když jsou návrhové vzory velice užitečné, počet který jich exituje je velký a některé se od sebe liší pouze maličkostmi. Toto stěžuje najít ten správný vzor na náš problém. Naštěstí se návrhové vzory rozdělují do několika kategorií.

Vzory pro tvorbu obejektů Skupina návrhových vzorů, která se stará o nejvhodnější vytváření objektů v systému. Příklad takových vzorů může být například Builder, Prototype, Abstract Factory nebo Object Pool, který implemetuje takzvaný thread pool o kterém jsem se párkrát zmiňoval.

Vzory týkající se struktůry programu Tyto vzory nám pomáhají tvořit jednotlivé vztahy mezi objekty, tak aby náš program byl lehce rozšířitelný a čitelný. Příkladem vzorů může být Decorator, Adapter, Facade, Composite a další.

Vzory týkající se chování Vzory, které pomáhají realizovat komunikaci mezi jednotlivými objekty a tím zvyšují flexibilitu celého systému. Příkladem může být Command, Observer, Strategy, Iterator a další.

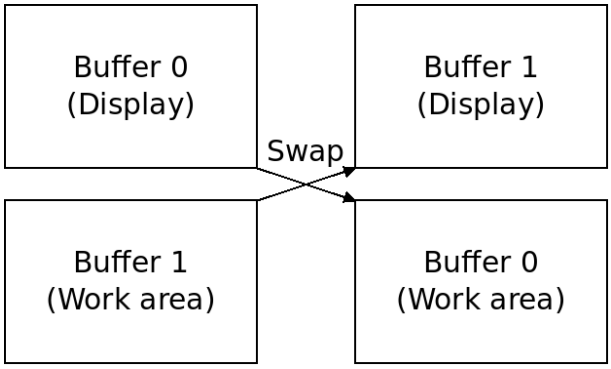
Nyní bych rád přistoupil k popisu jednotlivých návrhových vzorů, použité v této práci.

3.11.1 Double Buffer

Buffer nebo-li část stavu, který je modifikovatelný. Pokud tento buffer chceme inkrementálně upravovat, ale zároveň potřebujeme, aby okolní kód viděl pouze jednu atomickou změnu, můžeme použít tento návrhový vzor. Tohoto lze dosáhnout pomocí dvou instancí daného bufferu a to nynějšího a budoucího.

Pokud čteme informaci z bufferu, poskytujeme ji vždy z nynějšího bufferu. Pokud je informace zapisována do bufferu, zaspisujeme vždy do budoucího bufferu. Když máme všechny změny hotovy, musíme tyto dva buffery navzájem prohodit a změny na okolí budou působit instantně. Reprezentaci tohoto postupu můžeme vidět na obrázku 20. Přičemž starý nynější buffer může být znovu použit jako nový budoucí buffer. Použití tohoto návrhového vzoru je vhodné za těchto podmínek.

- Máme stav, který je modifikován inkrementálně.



Obrázek 20: Zobrazení dvojitého bufferu a jeho prohození[10]

- V případě, že by stav mohl být čten uprostřed modifikování.
- Nechceme aby okolní kód měl přístup ke stavu, který ještě není finální.
- Chceme číst stav, ale nechceme čekat než se provedou všechny modifikace.

Návrhový vzor také nese některé nevýhody. Hlavně se jedná o zdvojnásobení potřebné paměti pro reprezentaci daného stavu. Také je třeba zajistit, aby prohození bufferů bylo co nejrychlejší. Obecně se používá prohození referencí na daný stav, což je velice rychle. Občas se také můžeme setkat s variantou kdy se data překopírovávají.

Nejběžnějším příkladem je asi překreslování obrazovky, kdy se zobrazuje obsah jednoho bufferu zatímco se vykresluje obraz do druhého bufferu. Poté dojde k prohození a uživatel vidí na obrazovce nový obsah. Toto se však děje takovou rychlostí, kdy uživatel vidí pouze plynule změny na jeho obrazovce.

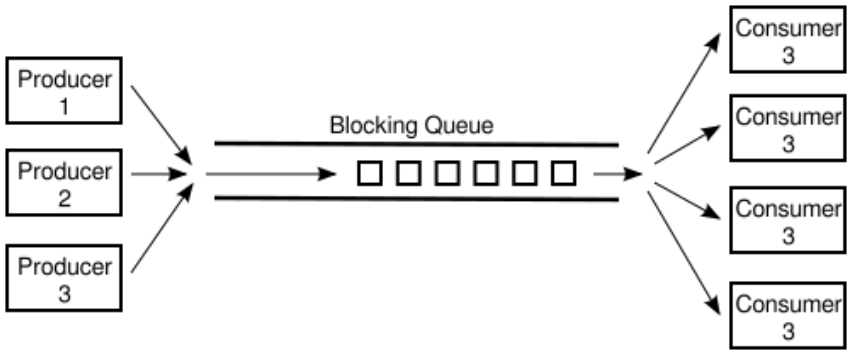
Já jsem danou metodu použil v momentu, kdy jsem potřeboval zajistit, aby si dvě vlákna vzájemně neměnila data.

3.11.2 Producer Consumer

Tento návrhový vzor je vhodný použít pokud chceme rodělit práci, která má být udělána a tím kdo danou práci udělá. Celý návrhový vzor se skládá ze tří hlavních částí.

- Někoho kdo vytváří jednotlivé úkoly nebo-li Producer.
- Fronta, do které Producer jednotlivé úkoly vkládá.
- Někdo kdo úkoly z fronty vytahuje a plní je nebo-li Consumer.

Separační část tvoří fronta, jelikož producer se poté co vloží úkol do fronty již nestará kdo úkol udělá a stejně tak consumer se nestará odkud se úkoly berou. Jediné co tyto dvě části mají společné je fronta. Fronta je většinou reprezentovaná nějakou blokující datovou struktůrou, což znamená že v jeden okamžik z ní může někdo buďto číst nebo do ní zapisovat. Dále je třeba



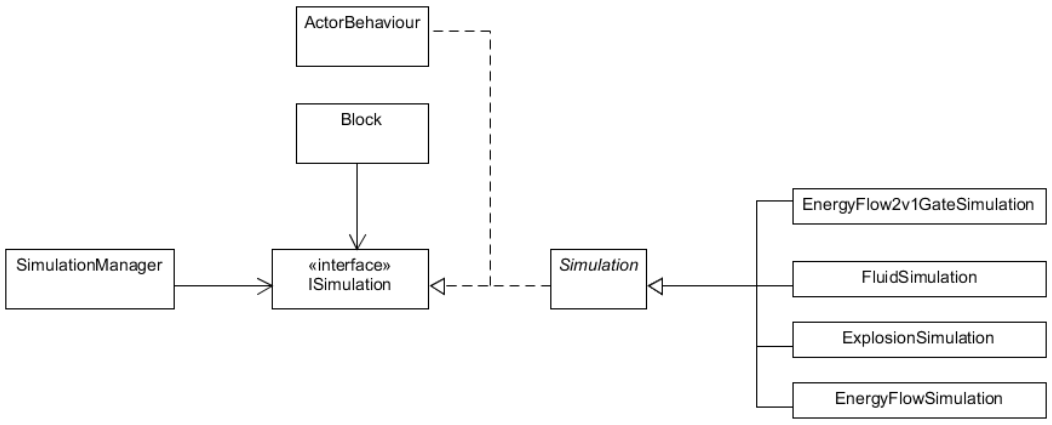
Obrázek 21: Funkce návrhového vzoru producer consumer[11]

podotknou, že do fronty může vkládat úkoly více producerů a plnit je může stejně tak více consumerů. Reprezentaci tohoto návrhového vzoru můžete vidět na obrázku 21.

Návrhový vzor je velice užitečný v případě několika vláken kde některé produkují úkoly a jiné tyto úkoly zpracovávají. Tohoto postupu jsem využil při aktualizaci chunků, kde jedno vlákno tvořilo požadavky na aktualizaci jednotlivých chunků a mezitím druhé vlákno chunky aktualizovalo. Více jak celý systém aktualizace chunků funguje můžete najít v kapitole [3.1.1] Aktualizace chunků.

3.11.3 Strategie

Návrhový vzor strategie je velice známý a hodně používaný. Patří do skupiny behaviorálních návrhových vzorů. V principu tento vzor rozděluje danou implementaci od rozhraní a veškeré závislosti ostatních tříd jsou směřovány na dané rozhraní, místo na specifickou implementaci. Tímto způsobem můžeme jednuduše dosáhnou dynamického chování v závislé třídě, protože za „běhu“ programu jsme schopni měnit implementaci daného rozhraní.



Obrázek 22: Použití návrhového vzoru strategie

Na obrázku 22 můžete vidět úryvek z uml návrhu systému simulací, který popisuji v kapitole [4.5] UML návrh. Lze na něm vidět jasně definované rozhraní ISimulation a dvě závislé třídy na tomto rozhraní. Specifické simulace sice dědí z abstraktní třídy Simulation, ale jelikož všechny závislosti jsou na rozhraní, mohl jsem do systému vcelku jednoduše přidat upravenou simulační třídu pro entity(ActorBehaviour), která poté může být předána simulačnímu manažeru. Simulační manažer nezajímá jak je dané rozhraní implemntováno, pouze jestli je typu ISimulation.

Tento návrhový vzor je také vhodný z hlediska testování, jelikož třídám poté můžeme podvrhnout testovací implementace. Například můžeme vytvořit testovací implementaci připojení do databáze, která se ve skutečnosti nebude do databáze připojovat, ale pouze vracet námi přesně nadefinované data podle testovacího scénáře.

4 Programátorská dokumentace

Dokumentace začíná obecným popisem celého projektu „Jiný Kosmos“, kde jsem popsal některé technologie, které projekt využívá a poté základní popis nejdůležitějších částí projektu. V dalších kapitolách jsem se poté zaměřil na popis práce s kódem simulací, jako například jak přidat novou simulaci nebo jaké užitečné metody lze využít při psaní vlastní simulační logiky. Zjednodušený návrh systému simulací poté můžete vidět v kapitole [4.5] UML návrh.

4.1 Obecný popis hry

V této části se zaměřím na obecný popis funkčnosti celé hry. Nikde nebudu zacházet do velkých detailů, ale pokusím se shrnout ty nejdůležitější části.

4.1.1 LibGDX

LibGDX je herní vývojový framework postavený na jazyce Java a pro vykrelování využívá OpenGL ES 2.0. Zaměřuje se na všechny dostupné platformy a na všech platformách se využívá jedno unifikované API[7]. Tímto LibGDX dosáhlo jednoduchého vývoje na různé platformy najednou. Vývojář nemusí psát několik různých variancí kódu, aby svou hru mohl vydat pro PC a mobilní zařízení. Mezi podporované platformy například patří tyto:

- Windows
- Linux
- Mac OS x
- Android (2.2+)
- BlackBerry
- iOS
- Java Applet
- Javascript/WebGL

Vzhledem k tomu, že naše aplikace je vyvýjena pouze na platformu PC, vidím pouze mírný nedostatek v použití OpenGL ES 2.0, které LibGDX používá a nelze změnit. Tato verze OpenGL je využita z důvody podpory mobliních zařízení a bohužel nenabízí všechny funkce jako nejnovější OpenGL 4+.

Dále potom framework nabízí spoustu užitečných funkcí, které při vývoji hry potřebujete.

Audio Streamování hudby, zvukové efekty pro WAV, MP3 a OGG.

Zpracování vstupů Pomocné třídy pro zpracování vstupu z myši, dotykové obrazovky, klávesnice, akcelerometr, kompas, dále také obsahuje třídu pro rozpoznání dotykových gest.

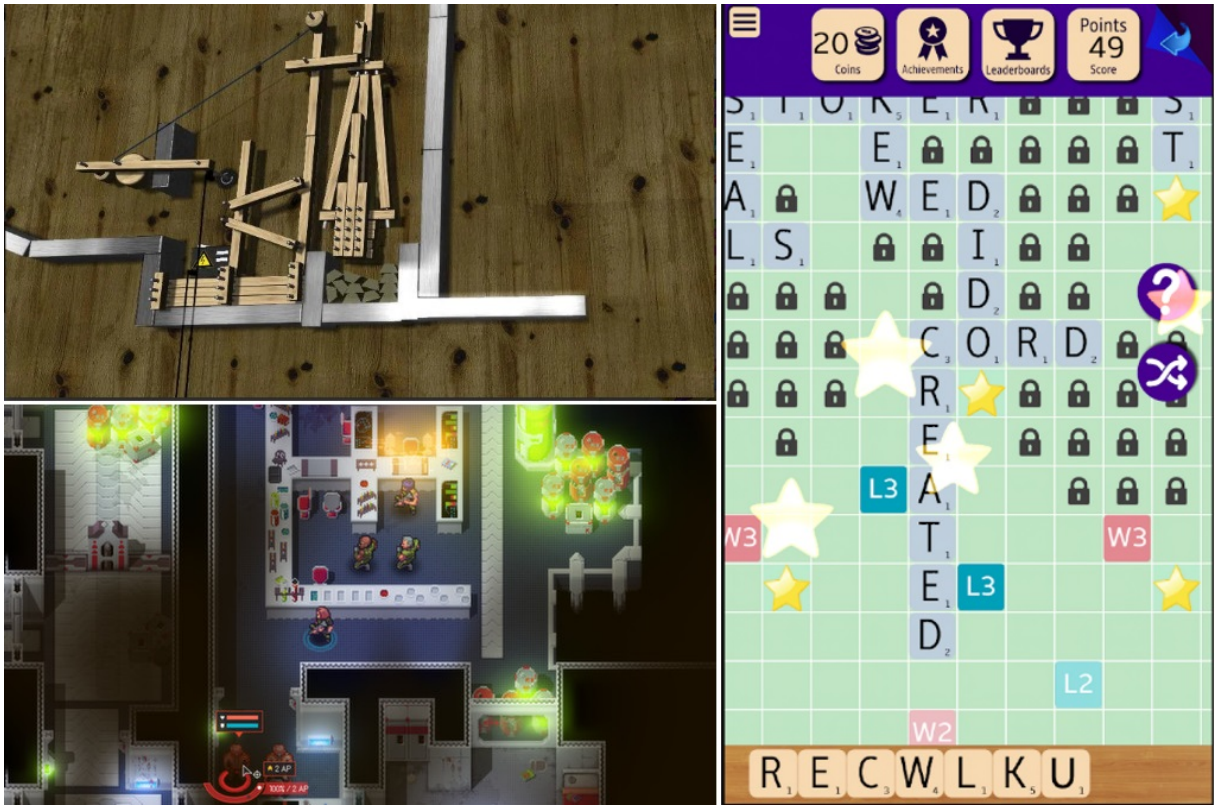
Matika a fyzika Pomocné třídy pro práci s maticemi, vektory nebo kvaterniony, pro obsluhu 2D fyziky využívá externí knihovny Box2D a pro 3D fyziku, využívá Bullet fyzikální framework.

Soubory a uložiště Podpora všech souborových systémů napříč platformami.

Grafika pro veškeré vykreslování využívá OpenGL ES 2.0 a nabízí tyto možnosti.

- Přístup přímo k pomocným metodám OpenGL. Jedná se o abstrakci například pro vertexové pole a buffery, delá nabízí možnost přímé práce s meshi, textůrami, shadery, framebuffery a další. Jedná se o způsob jak se dostat co nejblíž k rozhraní OpenGL a přeskočit veškerou režii od LibGDX.
- Vysokoúrovňové 2D rozhraní je určeno pro 2D hry a případné vykreslování GUI hry. Nabízí práci s orthografickou kamerou, bitmapami, textůrovými atlasy, fonty, 2D částicový systém a další.
- Vysokoúrovňové 3D rozhraní nabízí práci s perspektivní kamerou, nahrávání 3D modelu, 3D vykreslování s podporou materiálů a světelným systémem.

Tento rozsah možností použitých technologií je jeden z důvodů proč je LibGDX tak oblíbený framework. Důkazem je pár obrázků již vydaných her. Některé obrázky ze hry můžete vidět na obrázku 23.



Obrázek 23: Ukázky her používající framework LibGDX

4.1.2 Gradle

Gradle je nástroj pro automatizaci sestavování programů a správu závislostí. Díky tomuto nástroji můžeme jednoduše přejít na novější verze LibGDX a dalších externích knihoven. Konfigurační soubory se nachází v kořenovém adresáři a přesněji se jedná o soubor „build.gradle“. Tento soubor popisuje jak se má celý projekt zkompilovat a jednotlivé závislosti, které musí mít k dipozici. Dále je také možné najít stejně nazvaný konfigurační soubor ve složce „core“ případně „desktop“, které rozdělují celý projekt na podprojekty. Konfigurační soubory v těchto složkách se poté aplikují pouze při kompilaci daného podprojektu.

LibGDX používá tuto struktůru z důvodů rozdělení kódu pro jednotlivé platformy. V „core“ projektu se nachází všechny společné kódy pro všechny platformy, na které chceme naši hru vydat. Jedná se o 99% kódu. V dalších podprojektech jako je „desktop“ poté najdeme jednotlivé vstupní body pro danou platformu. Většinou se jedná o jednoduchou třídu nastavující specifické nastavení pro danou platformu a poté se předává kontrola do společného jádra hry.

4.1.3 Báliček block

Tento baliček obsahuje všechny kódy týkajících se práce s bloky. Můžeme zde najít třídu jako BasicBlockBundle, kde se registrují jednotlivé typy bloků do hry. Poté třídu BlockTypes, která

slouží jako datová struktůra pro list registorvaných bloků a také obsahuje některé pomocné metody pro získání ID nebo celé instance bloku. Nejdůležitější třídou v tomto balíčku je samozřejmě samotná třída Block. Tato třídá obsahuje jednotlivé vlastnosti, které můžou bloky nabývat jako jestli je blok průhledný, jestli umí uchovávat stav, jestli lze rozbít nebo ho můžeme držet. Také zde najdeme metody, které se volají v případě, že do bloku například udeříme nebo s ním chceme nějakým způsobem interagovat.

Nacházíse zde i několik podbalíčků jako „types“, který obsahuje implementace jednotlivých typů bloků, nebo balíček render, který obsahuje třídy určující jak se má daný blok vykreslit. Na některých obrázcích simulace vody jste si mohli všimnout zkosených hran, pokud vedle vodního bloku není žádný jiný blok. Právě toto zkosení se provádí v třídě DynamicBevelRender. Balíček state poté obsahuje třídy pro práci se stavem bloku.

4.1.4 Balíček map

Jeden z dalších velice důležitých balíčků, který obsahuje všechny třídy potřebné pro správu a vykreslování terénu. V této části bych však zmínil pouze tři nejdůležitější třídy a to Chunk, ChunkManager a World.

World Tato třída obsahuje metody pro změnu herního světa. Pokud chci ve světě něco změnit nebo přidat měl bych použít tuto třídu a né přistupovat přímo k ChunkManageru. Třída nabízí metody pro přidání hráče nebo jiné entity do světa, plánování simulací pro bloky, metody pro vykreslení entit a terénu, manipulaci a zjišťování informací o terénu.

ChunkManager Třída má nastarost všechny chunky, které reprezentují daný terén. Stará se nejen o existující chunky, ale taky umí tvořit nové nebo ničit nepotřebné a to i včetně nahrávání a ukládání na disk. Dále obsahuje hlavní aktualizační smyčku pro jednotlivé chunky, kterou jsem již popsal v kapitole [3.1.1] Aktualizace chunku.

Chunk Třída reprezentující jeden sloupek o velikosti 16x256x16 tvořen bloky. Obsahuje datovou struktůru bloků. Třída obsahuje metody pro práci s touto datovou struktůrou a také metody pro vygenerování terénu, přepočítání meshe, který tvoří viditelnou část chunku. Zde se také počítá propagace světla.

4.1.5 Třída Voxel

Tato třída je hlavním vstupem do programu a obsahuje hlavní vykreslovací smyčku v metodě render. Zde dochází k vykreslování jednotlivých částí hry jako je terén nebo entity. Taky zde dochází k základnímu nastavení všech subsystému hry jako je vytvoření světa, nastavení vykreslovacích shaderů nebo nastavení invetáře hráče.

4.2 Základní rozhraní simulačního objektu

Třída simulace, příhodně nazvaná Simulation implementuje rozhraní ISimulation a ISimulationResult. Rozhraní ISimulation obsahuje také definici výčtového typu Priority, určující prioritu dané simulace a časový interval, který určuje kdy se daná prioritní fronta musí odsimulovat. Rozhraní ISimulationResult rozšiřuje simulaci o její stav a možnost nastavit jestli daná simulace má pokračovat v dalším simulačnim cyklu nebo jestli se ukončí.

Dále třída Simulation obsahuje několik příhodných metod a hodnot, které lze využít v logice dané simulace.

- simulate je metoda, která musí být implementovaná rozšiřující třídou a jedná se o metodu, která by měla obsahovat logiku dané simulace
- cleanSimulationData je metoda volána při odstranění simulace ze simulačního procesu a měla by obsahovat uvolnění všech možných dat, které by mohly zahltit paměť RAM
- před každým spuštěním metody simulate se nastaví BlockProvider, který obsahuje blok dané simulace
- resetTime a resetTimeIfOver umí vyresetovat lokální čas běhu simulace resp. vyresetovat čas po uplynutí určené doby, vhodné pokud chceme například v dané simulační logice stopovat uběhnutí určité doby
- totalRunningTime je metoda vracející celkový čas běhu simulace
- metody onContinue a onRemove, které jsou volány simulačním manažerem v okamžiku kdy daná simulace pokračuje dalším simulačním cyklem nebo při jejím odstranění

4.3 Nová simulace a propojení s blokem

K tomu, aby blok mohl být simulován je zapotřebí aby jeho třída implementovala rozhraní IBehaviour, které definuje pouze jednu metodu a to createSimulation. Tato metoda vrací instanci typu ISimulation. Jedna z možností je vrátit novou instanci anonymní třídy Simulation, kde implementace samotné simulace bude svázána s daným blokem. Druhou možností je vytvořit nový typ simulace v balíčku simulation/types a poté vracet kokrétní instaci dané třídy. Také lze využít kombinace těchto dvou přístupů, kdy vytvoříme nový abstraktní typ simulace a pomocí anonymní třídy v daném bloku pouze specifikujeme simulaci pro daný blok.

Tohoto postupu jsem například využil pro vytvoření simulace AND a OR logického hradla, kde oba hradla mají stejný počet vstupů i výstupu, jediné v čem se liší je jejich vnitřní logika. Proto jsem vytvořil abstraktní třídu EnergyFlow2v1GateSimulation, kde 2v1 říka, že jde o typ hradla se dvěmi vstupy a jedním výstupem. Tato třída má metodu gateLogic, která přijímá 4 parametry a vrací jestli je dané hradlo otevřené nebo zavřené. První dva parametry říkají jestli blok má připojen nějaký vstup a druhé dva parametry určují jaká je hladina energie na daných

vstupech. Potom už stačí, aby naše bloky AndBlock a OrBlock vracely instanci této třídy s jinou implementací metody gateLogic.

4.4 Využití časových simulací

Pokud chceme, aby daný blok měl časovanou simulaci, je zapotřebí aby metoda createSimulation nevracela instanci třídy Simulation, ale instanci TimedSimulation. Tato třída rozšiřuje základní třídu Simulation a navíc implementuje rozhrani ITimedSimulation, které přidává několik potřebných metod.

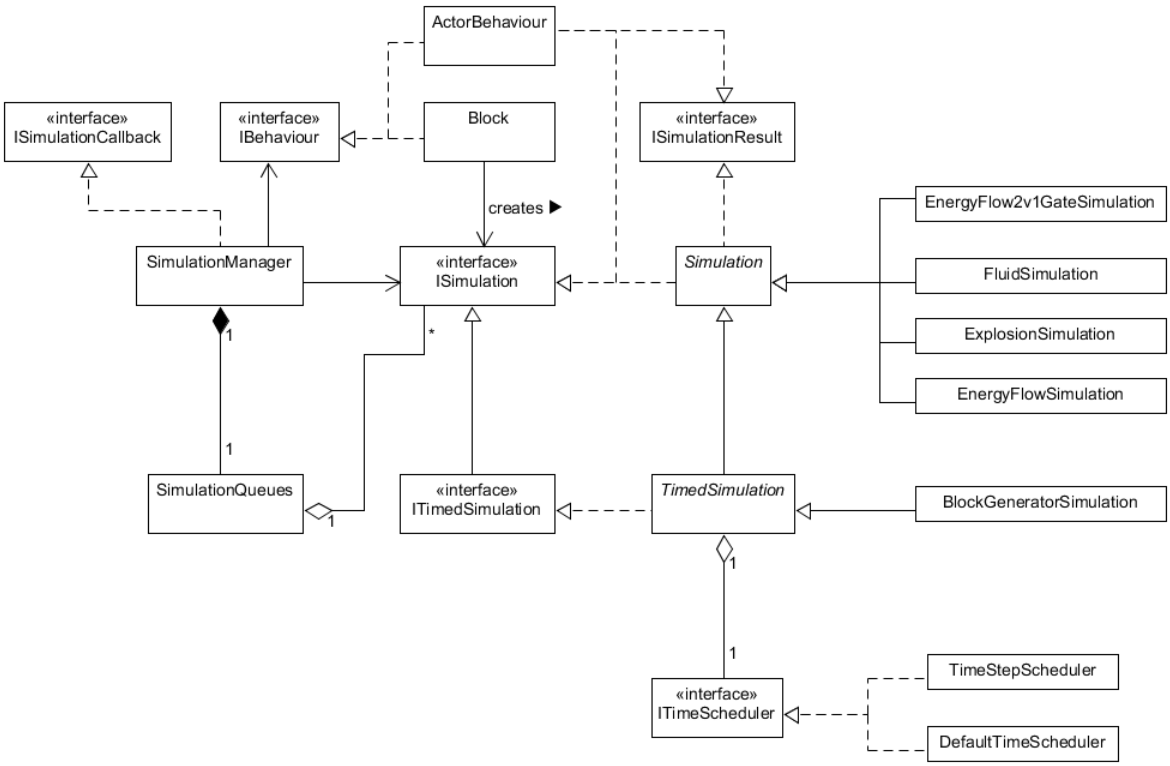
- notifikační metody jako onSleep, onSleeping a onWakeup, kde první metoda se zavolá jednou v okamžiku kdy se daná simulace uspí, druhá metoda se volá v každém simulačním cyklu pokud simulace je uspaná a třetí se zavolá v momentě, kdy dojde k probuzení simulace
- getMaximalLifetime, metoda vrací maximální dobu provádění simulace, doba se uvádí v milisekundách
- metody sleep a awake, slouží pro potřeby simulačního manažeru, který se na základě výstupu těchto metod, rozhodne jestli danou simulaci má uspat nebo probudit
- timeScheduler, jedná se o instanci rozhraní ITimeScheduler, který je zodpovědný za poskytnutí logiky, kdy se daná simulace má uspat nebo probudit.

4.5 UML návrh

Návrh celého systému simulací můžete vidět na obrázku 24. Jedná se o zjednodušený návrh bez znázorněných metod a instančních proměnných. Obrázek má sloužit jen pro získání obecného přehledu jak jsou jednotlivé třídy propojené. Jak jednotlivé části fungují jsem již popsal v předešlých kapitolách. Zopakuji jen to nejdůležitější. Simulační manažer je řidící logikou celého simulačního procesu. Obsahuje simulační fronty a logiku spouštění jednotlivých simulací. Také reaguje na výsledek jednotlivých simulací a podle toho danou simulaci odstraní nebo jí zařadí zpět do simulační fronty.

Samotnou simulaci představuje třída Simulation nebo její variace TimedSimulation, z těchto tříd jsou poté odvozeny jednotlivé typy simulací jako FluidSimulation pro tekoucí vodu nebo ExplosionSimulation pro simulaci výbuchu TNT. TimedSimulation navíc obsahuje instaci TimeScheduler, která rozhoduje jakým způsobem se daná simulace bude uspávat a probouzet.

Celého tohoto systému poté využívá třída Block a ActorBehaviour, které jsou zodpovědné za vytvoření instance simulace. Rozdíl mezi Block a ActorBehaviour je ten, že ActorBehaviour již je samotnou instancí simulace a tím pádem obsahuje danou simulační logiku. Block na druhou stranu pouze vytváří některou z typů simulace, podle toho o jaký blok se jedná.



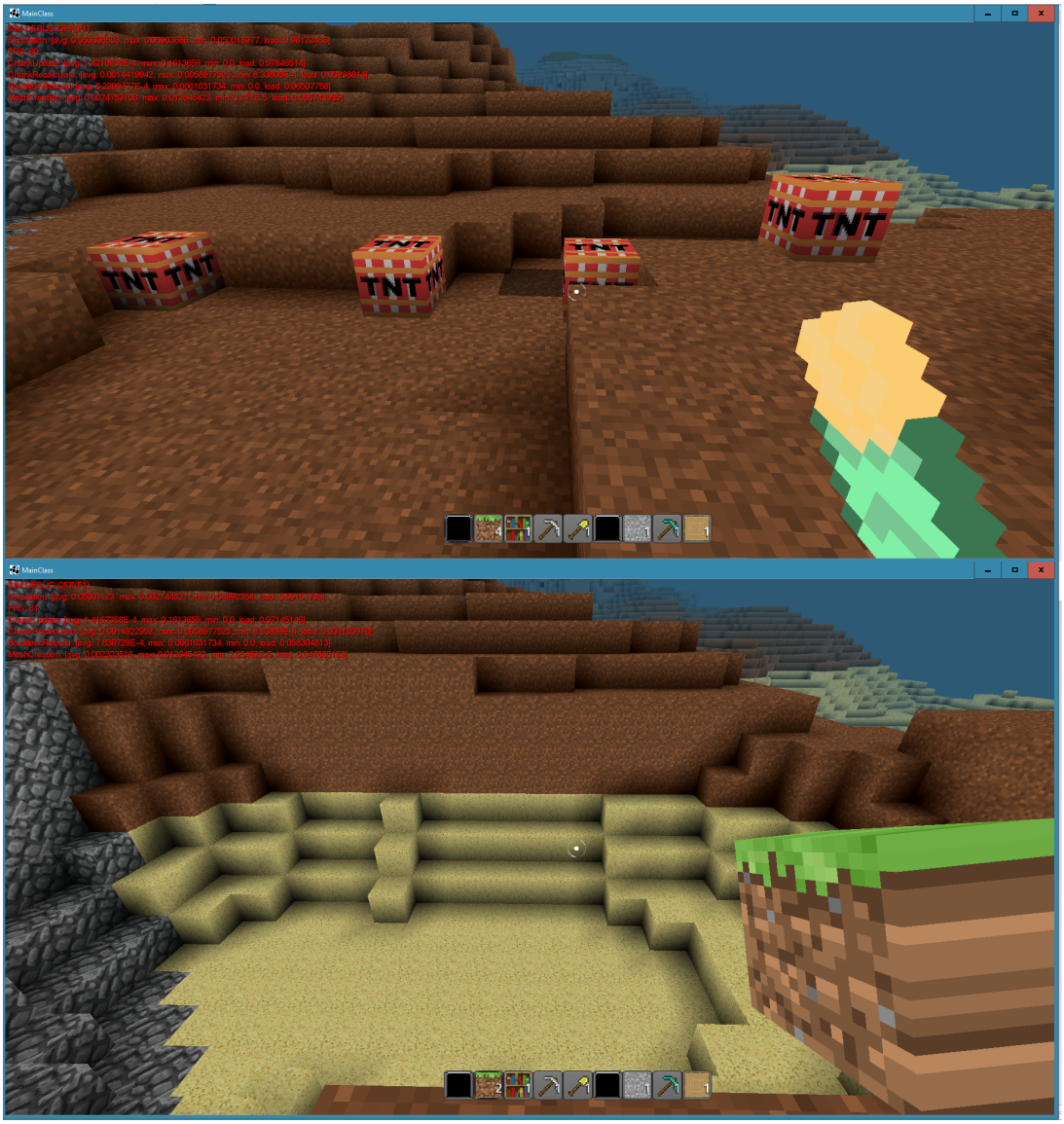
Obrázek 24: Návrh celého systému simulací

5 Uživatelská příručka

Pro vyzkoušení a shlédnutí simulace jsem nastavil hru tak, aby každý hráč v invetáři měl potřebné bloky. Po nahrání terénu a kliknutí na klávesu „I“ se zobrazí GUI invetáře. Zde můžete vidět jaké bloky máte u sebe, aby bylo možné bloky použít je třeba na ně najet myší a přetáhnout na lištu rychlého použití. Po přetažení jednotlivých bloků můžete invetář zavřít opětovným kliknutím na klávesu „I“. Poté je již možné si pomocí kolečka na myši vybrat daný blok, který chceme umístit do hry. Levým tlačítkem myši bloky odstraňujete, na každý blok je třeba kliknout třikrát, aby se odstranil a přidal do vašeho invetáře. Pravým tlačítkem lze přidat blok, který zrovna držíme v ruce. Kam se blok vloží určuje označený blok. Toto označení se vždy přichycuje k již existujícím blokům a tím pádem lze pouze přilepit blok k nějakému jinému. Pokud bychom chtěli blok přidat do volného prostoru, podržíme levý ALT a kolečkem myši ovládáme vzdálenost označeného bloku od nás. Po výběru stačí kliknou pravým tlačítkem myši a blok se na dané místo umístí.

V invetáři je nachystáno několik typů bloků a to. EarthBlock, který generuje vodní bloky co 5 sekund. WaterBlock, jedná se o vodní blok, který když vložíme do světa, začne se rozlévat a poté se vypaří. TNTBlock, typ bloku, který umí zničit bloky kolem sebe a GrassBlock, který slouží jako spouštěč detonace TNTBlocku. Aby TNT vybuchlo je potřeba aby se jednou ze stran dotýkal právě zmíněného GrassBlocku. Pokud tato situace nastala, zhruba po 3 sekundách dojde k explozi a zničení okolních bloků. Pokud v dosahu exploze je jiné TNT, poté toto TNT také exploduje. Prezentaci výbuchu TNT můžete vidět na obrázku 25. Dále je možné si vyzkoušet větší simulaci vody, kdy stačí odebrat pevný blok na pobřeží nějaké vodní plochy.

Další ukázkou simulací, jsou entity. Vygenerovat entity lze podržením levého SHIFTU a kliknutím na levé tlačítko myši. V tomto okamžiku se vygenerují čtyři entity na místě, které máte označeno. Tyto entity vás poté budou následovat.



Obrázek 25: Obrázek před a po výbuchu TNT

6 Závěr

Práce na tomto projektu byla obohacující zkušenost, jelikož toto bylo poprvé co jsem programoval jakoukoliv 3D hru. Nabyl jsem různé znalosti od vykreslování 3D grafiky až po programování vícevláknové aplikace. Dále jsem byl nucen při vývoji přemýšlet jaký efekt budou mít dané úpravy na výkon celé aplikace a paměť. Výsledkem mého snažení je funkční systém simulací, který lze jednoduše obohatit o další typy simulací, aniž by kladl velké nároky na programátora.

Vše však není úplně jak bych si představoval. Plynulost simulace při velkém množsví simulovaných bloků je sice v pořádku, ale vykreslování již nevypadá tak pěkně jak bych chtěl. Vykreslování změn simulací byl jeden z největších problémů a bohužel se mi ho nepodařilo vyřešit úplně dokonale. Toto je jeden z hlavních bodů, které by v budoucích verzích projektu Jiný Kosmos měly být zlepšeny.

Literatura

[1] BANKS, Jerry. Discrete-event system simulation [online]. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2001 [cit. 2018-04-29]. ISBN 01-308-8702-1.

[2] Continuous Simulation [online]. [cit. 2018-04-29]. Dostupné z: https://web.archive.org/web/20110609000757/http://www.cs.uu.nl/docs/vakken/sim/continuous.pdf

[3] Introduction to Discrete-Event Simulation and the SimPy Language [online]. s. 33 [cit. 2018-04-29]. Dostupné z: Introduction to Discrete-Event Simulation and the SimPy Language

[4] Discrete event simulation (DES). WhatIs.com [online]. 2012 [cit. 2018-04-29]. Dostupné z: https://whatis.techtarget.com/definition/discrete-event-simulation-DES

[5] Step function [online]. In: . [cit. 2018-04-29]. Dostupné z: https://stackoverflow.com/questions/8988871/plotting-a-step-function-in-mathematica

[6] Cellular Automata. The Nature of code [online]. [cit. 2018-04-29]. Dostupné z: http://natureofcode.com/book/chapter-7-cellular-automata

[7] Libgdx. Libgdx [online]. [cit. 2018-04-29]. Dostupné z: https://libgdx.badlogicgames.com/features.html

[8] Factorio. Factorio [online]. [cit. 2018-04-29]. Dostupné z: https://www.factorio.com/screenshots

[9] Kingdom Come: Deliverence [online]. [cit. 2018-04-29]. Dostupné z: https://www.kingdomcomerpg.com/cs

[10] GBA Graphics Programming. University of Mary Washington [online]. [cit. 2018-04-29]. Dostupné z: http://cs.umw.edu/ finlayson/class/fall17/cpsc305/notes/08-graphics.html

[11] Threads and Parallel Processing [online]. [cit. 2018-04-29]. Dostupné z: http://math.hws.edu/javanotes/c12/s3.html

[12] Weather forecast. Offshore technology [online]. [cit. 2018-04-29]. Dostupné z: https://www.offshore-technology.com/features/featureworried-about-the-weather-using-mathematical-modelling-to-reduce-offshore-risk-4378904/

[13] 3 Computer Simulations that Changed The World. Gizmodo [online]. [cit. 2018-04-29]. Dostupné z: https://io9.gizmodo.com/5923950/3-computer-simulations-that-changed-the-world-and-2-that-are-on-the-verge

7 Seznam příloh

Obsah přiloženého CD

\src\core - Zdrojové kódy aplikace

\src\desktop - Zdrojové kódy pro platformu PC

\bin - Soubor pro spuštění aplikace na PC

\CEC0080.pdf - Diplomová práce ve formátu PDF